UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial :

Présentée par

Selma Saïdi

Thèse dirigée par **Dr. Oded Maler**

préparée au sein **du laboratoire VERIMAG**
et de **l'école doctorale Mathematiques, Sciences et Technologies de l'Information, Informatique (MSTII)**

Optimizing DMA Data Transfers for Embedded Multi-Cores

Thèse soutenue publiquement le **24 Octobre 2012**,
devant le jury composé de :

Mr, Ahmed Bouajjani
Professeur à l 'université Paris diderot, Président
Mr, Luca Benini
Professeur à l 'université de Bologne, Rapporteur
Mr, Albert Cohen
Directeur de Recherche, INRIA, Rapporteur
Mr, Oded Maler
Directeur de Recherche, CNRS, Directeur de thèse
Mr, Eric Flamand
Directeur de la division AST-Computing, STMicroelectronics, Examinateur
Mr, Bruno Jego
Equipe Application à AST-computing,STMicroelectronics, Examinateur

Contents

Introduction

This thesis has been conducted as part of a CIFRE contract with ST and in the framework of the collaborative project ATHOLE with the participation of ST, CEA-LETI, Verimag, Thales, and CWS. The project was focused on *embedded multi-core architectures* developed by ST (and CEA-LETI), initially xSTream and then Platform 2012 (P2012). The role of Verimag in the project was to investigate ways to ease the passage from hardware implementation of key applications (such as video and signal processing) to parallel software. Exploiting the parallelism offered by multi-cores is an industry-wide and world-wide challenge, and this thesis demonstrates what can be done for a specific (but very general) class of applications and a specific class of execution platforms where the multi-core is viewed as a computation fabric to which the main processor delegates heavy computations to execute in parallel.

The applications that we consider are those that would be written as a (possibly nested) loop in which the same computation is applied to each and every element of an array of one or two dimensions. Such applications are called *data parallel* or *embarrassingly parallel* and they can be found in image and video processing or in scientific computations. Running such an application efficiently on a multi-core architecture involves a lot of decisions that require acquaintance with the low-level details of the architecture. These decisions involve the splitting of the data into pieces that are sent to the different processors, scheduling competing tasks and selecting efficient communication and data transfer mechanisms. Today such decisions are made manually, and they pose non-trivial combinatorial puzzles to application programmers and reduce their productivity.

The goal of this thesis is to liberate, in the limits of the possible, application programmers from this task by automating it, either fully or partially. The ideal scenario envisioned by this thesis is that the programmer writes his or her algorithm as a generic iteration of the computation over the array elements, annotates it with some performance related numbers (execution time for processing one array element, amount of data involved), and based on an abstract model of the architecture (processor speeds, interconnect bandwidth and latency, memory features) *automatically* derives a parallel execution scheme which is optimal or at least acceptable with respect to timing and/or other performance metrics. In other words, we frame the problem of optimal execution as an optimization problem based on application and architecture parameters.

We focus on architectures where the cores use a fast but small scratchpad memory and the main issue is how to orchestrate computations and data transfers from off-chip memory to the local memory of the cores efficiently. We use a double buffering scheme where optimization reduces to the choice of the size and shape of blocks which are subject to a single DMA call. This choice is dependent, of course, on the cost model of the DMA and the interconnect, as well as on features of the application such as the ratio between computation and communication and the weight of data sharing between blocks.

This thesis ran in parallel with the design of P2012, a fact that prevented us from validating our analysis on this architecture. We chose the Cell B.E, a mature architecture whose features are close to P2012. We are currently extending our work to P2012 where the DMA is more centralized and the local memory is *shared* between all the cores that reside in the same cluster.

The rest of the thesis is organized as follows.

– Chapter 2 is a survey of current trends in MPSoCs (multi-processor systems on chip) and the industrial context of the thesis;
– Chapter 3 gives preliminary definitions of the hardware and software models. It explains DMA in general and the one implemented in the Cell B.E architecture that we use for benchmarking. On the software side it explains the structure of data parallel applications, the different ways to partition one and two-dimensional data, and describes the well-known double buffering scheme which allows to process one block of data while fetching the next block in parallel. The problem of optimal granularity is formulated;
– Chapter 4 solves the optimal granularity problem for one-dimensional and two-dimensional data arrays. The solution is based on the analysis of the behavior of double-buffering software pipeline which depends on the computation/communication ratio of the basic computation as well as on the choice of granularity.
– Chapter 5 extends the analysis to computations where data is shared and the computation for an array element involves some data belonging to its neighbors. For one-dimensional data arrays we compare three strategies for data sharing, namely replication of data in DMA calls, inter-process communication and load/store instructions in the local memory. For two-dimensional data our model captures the tension between memory layout constraints that favor flat blocks and data-sharing considerations that favor square data blocks;
– Chapter 6 is dedicated to the validation of the theoretical results on a cycle-accurate simulator of the Cell B.E;
– Chapter 7 concludes the thesis with suggestions for further work, including adaptation to the specifics of the P2012 architecture.

Chapter 1

Embedded Multicores: Opportunities and challenges

1.1 Embedded Multicore Architectures

Multicore architectures are a reality in most products today. Graphical Processing Units (GPUs) [OHL+08] are perhaps the most widely visible example of this trend featuring hundreds of cores and have highly contributed to the excitement about multi-cores because of the high perfomance numbers they exhibit.

In fact, multi-core platforms became *the* alternative to increase performance in a system after the microprocessor industry has hit the *power wall* (also referred to as the speed wall) preventing higher clock speeds. Indeed, Moore's law predicted 40 years ago that the number of transistors on a chip roughly doubles every two years increasing the ability to integrate more capabilities onto one chip, and thereby increasing performance and functionality and decreasing the cost. However, doubling performance per processing element, which is the main feature Moore's law is based on, is not feasible anymore because faster processors also run hotter. Therefore running 2 processors in the same chip at half the speed is less energy consuming and potentially equally performance efficient.

In particular, *embedded* multi-core architectures [1] have known a major wave of interest in the past years with the growing demand for integrating more functionalities in embedded devices, smart phones being a prime example. Due to the rapid advances in the silicon industry, *Multiprocessor Systems on Chips* are becoming an important feature of embedded systems.

In the sequel, we present some important features of MPSoCs and describe Platform 2012, a many-core programmable system on chip developed jointly by STMicroelectronics and CEA. We then present and discuss some memory issues related to such platforms which constitute the main focus of this thesis.

1.1.1 Multiprocessor Systems on chip (MPSoCs)

The term Systems on Chip (SoC) refers to embedded systems integrating in a single chip several hardware modules such as processing units, memories and vendor specific Intellectual Properties (IPs), designed and optimized for a specific class of applications

1. Also referred to as Chip Multiprocessors (CMP).

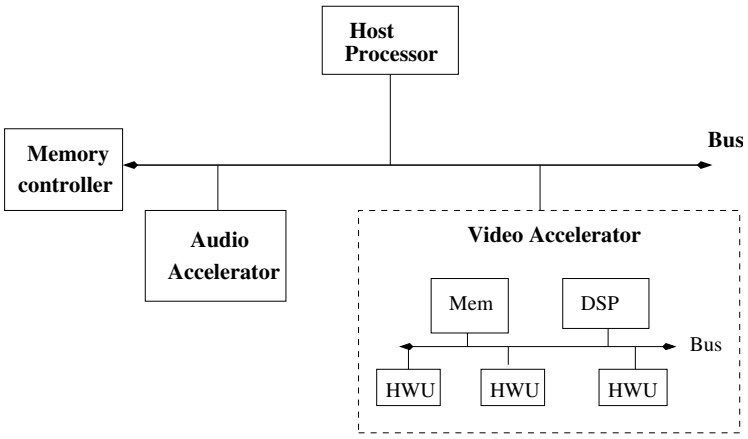


Figure 1.1: Overview of the STNomadik platform.

such as networking, communication, signal processing and multimedia. Today the semiconductor industry is mainly driven by these applications markets justifying the cost of chips design which represents tens of millions of dollars.

The embedded feature of these architectures makes them resource constrained with a strong concern for cost and power efficiency. Their application specific nature can then be used to tailor the system architecture to suit the needs of the target application domain and meet the strict area and power budgets. For instance, if the target applications do not use floating point arithmetic then no floating point unit is considered in the design of the SoC.

Multiprocessor Systems on Chips (MPSoCs) [WJM08] usually integrate a powerful host processor combined with a mixture of specialized co-processors such as Digital Signal Processors (DSPs), and dedicated hardware units that implement in hardwired logic computationally intensive kernels of code. The Nomadik architecture [ADC+03] developed by STMicroelectronics is an example of such platforms. Figure 1.1 presents a simplified overview of the STNomadik platform. It is designed for mobile multimedia applications and composed of a main processor and *application-specific accelerators* for audio and video, all connected through a bus. The main processor focuses on coordination and control as most of the multimedia functions are performed by the accelerators which are in turns heterogeneous subsystems composed of a multimedia DSP and dedicated hardware units (HWU) that implement several important stages of video processing.

Today, embedded applications are rapidly evolving and growing in complexity, as an example the H264 video compression standard reference code consists of over 1200000 lines of C code. Hence, they are increasingly presenting conflicting requirements of both high performance and low power consumption. Hardwired implementations of such applications have the advantage of being very efficient, however they have a clear limitation: flexibility. Therefore, today the design trend of MPSoC platforms is becoming highly *programmable* to replace specialized hardware by multi-core subsystems, as depicted in Figure 1.2. Unlike general purpose multi-core systems, multi-core accelerators use a *large* number of low to medium performance cores. The memory on the accelerator part is referred to as *on-chip memory* and the host memory is usually referred to as main memory or *off-chip memory* when it is located externally. Obviously there is a difficult design decision as what to implement in software and hardware in order to find the best

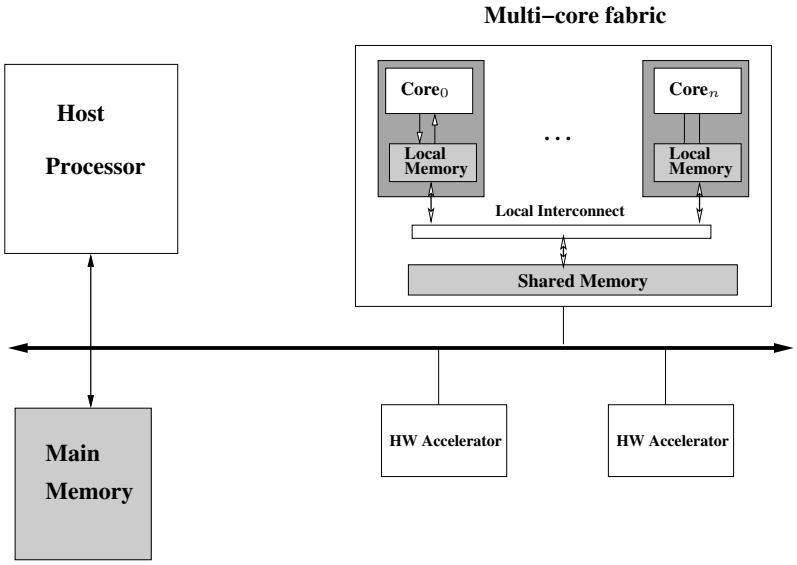


Figure 1.2: A Multiprocessor System on Chip.

trade-off between efficiency and flexibility.

In the last past years, the following important features have characterized the evolution of MPSoCs design:

- The integration of *Network on chips* (NoCs) technology [BM02] as a new paradigm for SoC communication infrastructures. The principles of NoCs are borrowed from packet based communication networks and have been proposed mainly to encounter the scalability limits of buses and switches as more and more hardware components are being integrated into a single chip. Connecting these components efficiently is a major concern for both performance and energy.
- The adoption of *globally asynchronous and locally synchronous* (GALS) paradigm [IM02] which integrates different clock and frequency domains into a single chip. By connecting locally synchronous modules via asynchronous wrappers, GALS circuits encounter the problem of controlling the global clock signal propagation across the entire chip which deeply affects power consumption.
- The adoption of *scratchpad memories* [BSL$^+$02] as a design alternative for on-chip memory caches. A scratchpad memory occupies the same level of the memory hierarchy as a cache and constitues, with the main memory, a global address space. Unlike caches, data movements are managed by the software giving more control over the access times and more predictability to the program.

This draws the big picture of the context of this thesis which focuses on embedded multi-core platforms that constitute *non-autonomous* systems and are rather used as *general purpose accelerators* to guarantee high performance and flexibility. They are coupled with a host processor which runs the operating system and offloads computationally heavy and parallelizable kernels of code to the multi-core fabric to be accelerated via parallel execution, somewhat similar to GPUs.

The most significant difference between a host only program and a host+accelerator program is that the on-chip memory may be completely separated from main memory

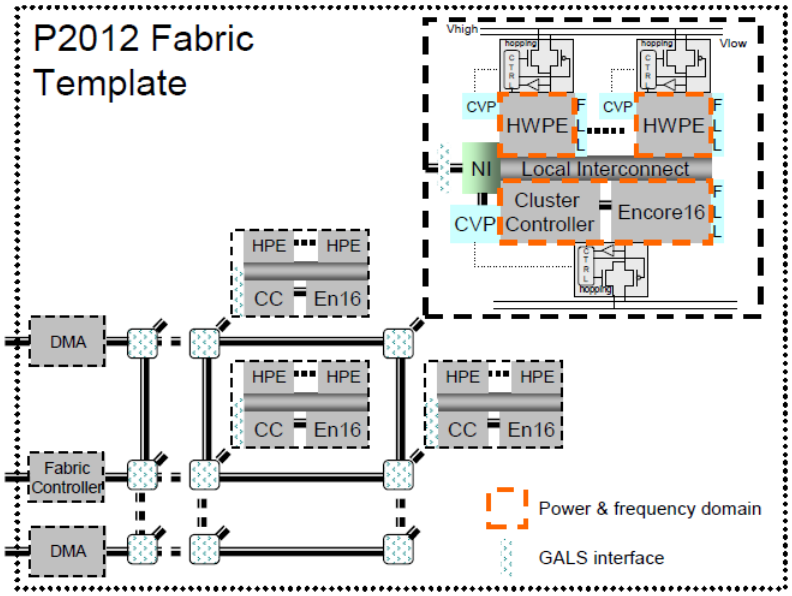


Figure 1.3: Platform 2012.

(which is the case when scratchpad memories are used). In this case, the cores may not be able to read or write memory data directly and all data movements between the separate memories must be performed explicitly, typically using a *Direct Memory Access* (DMA) engine. In the following, we present Platform 2012, an MPSoC architecture having this feature:

1.1.2 Example of an MPSoC: P2012

Platform 2012 (P2012) [SC10] is an area and power efficient many-core computing fabric intended to *accelerate* applications such as video decoding, imaging and next generation immersive applications like computational photography and augmented reality. The ultimate goal of P2012 is to fill the area and power efficiency gap, explained earlier, between general-purpose embedded CPUs and fully hardwired application accelerators, the former being more flexible and the latter more efficient.

Figure 1.3 presents an overview of the platform. It is highly modular as it is composed of a number of clusters where each cluster has its own frequency and clock domain and is capable of integrating hardwired accelerators. Clusters are connected through a high performance fully Asynchronous Network On Chip (ANoC) organized in a 2D-mesh structure providing a scalable and robust communication across the different power and clock domains.

A cluster, called ENCore, is a multi-core processing engine clustering up to 16 processors. Each processor is an STxP70-V4 which is an extensible 32-bit RISC processor core implemented with a 7-stages pipeline, reaching 600 MHz and it can execute up to two instructions per clock cycle (dual issue). P2012 can also link to the ENCore cluster a set of hardware processing elements (HWPEs) that provide a cost optimized code implementation when a software implementation is inefficient.

The memory hierarchy in the fabric consists of 3 levels, the first intra-cluster level which is a multi-banked one cycle access L1 data memory shared between processors in

the same cluster, a second inter-cluster level shared between clusters in the fabric and a third off-chip memory level shared between the fabric and the rest of the SoC components. The on-chip memories are scratchpad memories and constitute with the off-chip memory a visible global memory map with a Non Uniform Memory Access (NUMA) time.

In this platform, remote memories (off-cluster and off-fabric) are very expensive to access making off-chip memory access the *main* bottleneck for performance. Direct Memory Access (DMA) engines are available per cluster to guarantee hardware accelerated memory transfers. Their efficient usage is delegated to the software/programmer who becomes responsible of making decisions about data granularity, the partitioning and the scheduling of data transfers. This issue along with others memory related issues are detailed and discussed in the next section.

1.1.3 Memory Organization

One of the most critical components that determine the success of an MPSoC based architecture is its memory system [WJM08]. Whereas for conventional architectures caches are an obvious choice, in MPSoCs several memory design configurations can be considered using caches, scratchpad memories, stream buffers or a combination of those.

Despite the continuous technology improvement for building larger and faster memories, accessing main memory still remains the bottleneck for performance in many hardware systems. In MPSoCs this limitation is even more critical mainly because i) the limited on-chip memory capacity, especially compared to the increasing requirement of handling larger data sets needed by embedded applications, ii) the main memory is shared between the host processor and other devices and IPs on the SoC.

Over the decades, the gap in the increase of performance between processors speed and memory speed has been referred to as the well known *memory wall* [WM95]. The system performance becomes then determined by memory access speed rather than the processor's speed. This gap has grown over the years to reach a factor of 100 today as depicted in Figure 1.4. This is a big issue in computer architectures since in most programs 20 to 40 % of instructions are referencing data [HP06]. This fact is more significant for data intensive applications that constitute a large part of today's applications.

Memory hierarchy has been proposed as a solution to reduce the memory gap by providing different levels of memories with increasing speed and decreasing capacity, as illustrated in Figure 1.5, where main memory constitutes the last memory level in the memory hierarchy as it is the first location of input data and where the output data will eventually reside [2]. Traditionally other memory levels are referred to as *caches*.

Cache Based Architectures

A cache memory keeps a *temporary* view of a small part of the main memory so that the access to a data item is very fast and referred to as a *cache hit*, if a copy of this data item is in a first level cache. The access is much slower and is referred to as a *cache miss* if the copy is located in a further cache level or in main memory. When a cache miss occurs, a *fixed size* collection of data items containing the requested word called a block or a line is retrieved from main memory and placed into the cache. Failing to keep pace with the processors speed, the main attempt was to reduce the number of cache

2. Secondary storage such as disks are ignored.

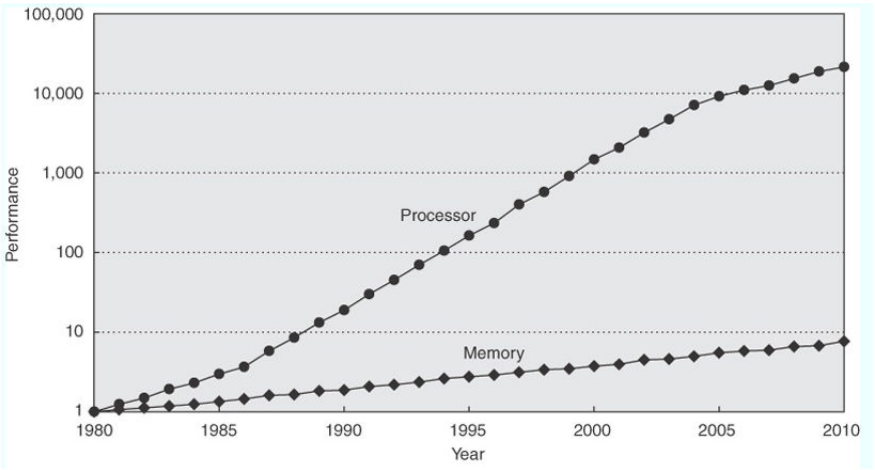


Figure 1.4: The gap in performance between processor speed and memory speed.

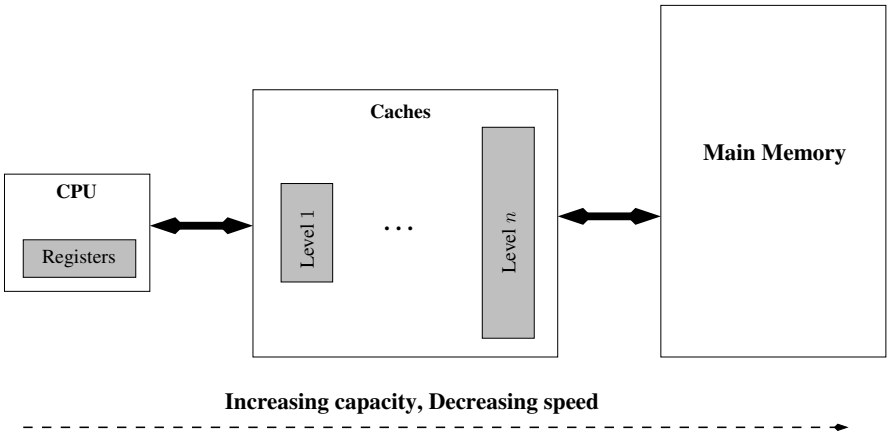


Figure 1.5: Memory Hierarchy.

misses by exploiting the temporal and spatial locality of programs. Temporal locality exploits the fact that the same data item will be needed by the program in the near future whereas spatial locality the fact that another data item in the same block will be most likely needed soon. Efficient cache management policy has been (and is still) an active research area [WL91, MCT96].

In Shared Memory symmetric Multiprocessor (SMP) context, cache solutions suffer from the *cache coherence* (consistency) problem where two processors or more may access in their private cache different copies of the same data. Therefore, the memory view can easily get inconsistent if one of the processors modifies the value of the cached data but not the external copies. The cache has to be flushed to upgrade the data value in main memory, or the cached value has to be invalidated. To manage the consistency issue between caches and memory, many solutions have been proposed ranging from hardware to software solutions [LLG$^+$90, AB86, Ste90], hardware based solutions being the most popular. Non coherent systems leave the management of cache inconsistency to the software.

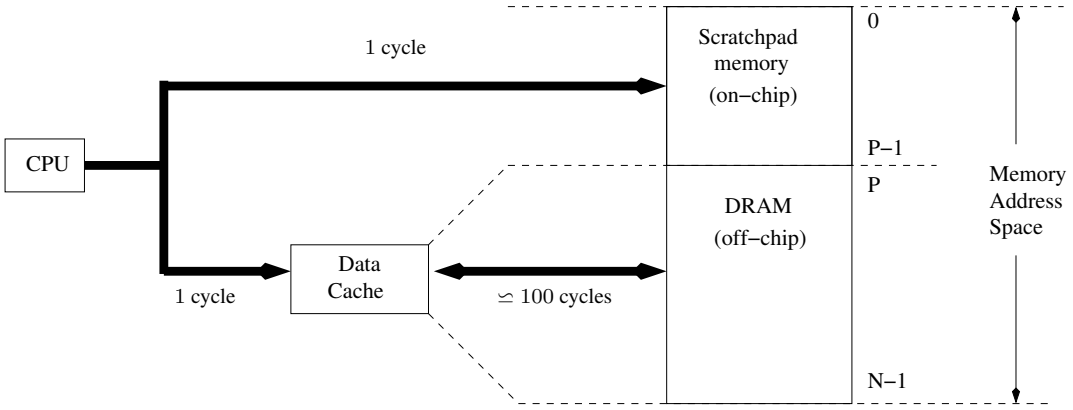


Figure 1.6: Dividing data address space between SPM and off-chip memory.

While for general purpose multiprocessor architectures, the design choice of using caches is obvious, it is not the case anymore for MPSoCs. The main reason is that MPSoCs are resource constrained systems and managing cache coherence is very expensive both in terms of area and power [LPB06]. Therefore Scratchpad Memories (SPMs) have been proposed as an alternative to caches where data (and sometimes code) transfers through the memory hierarchy are *explicitly* managed by the software. Furthermore, unlike a general purpose processor where a standard cache-based memory hierarchy is employed to guarantee good average performance over a wide range of applications, in MPSoCs, the overall memory organization has to be tailored in order to fit at best the needs of the target class of applications. Indeed, according to the memory access pattern of the target applications, the use of scratchpad memories can be more efficient than caches [SK08].

Explicitly Managed Memory

Both caches and SPMs allow fast access to the residing data, whereas an access to the off-chip memory requires relatively longer access times. The main difference between a scratchpad memory and a conventional cache is that SPMs are mapped into an address space *disjoint* from that of the off-chip memory as illustrated in Figure 1.6. Moreover, a SPM guarantees a *fixed* single-cycle (for first level) access time whereas an access to the cache is subject to cache misses. Therefore to improve performance, frequently accessed data can be directly/statically mapped to the SPM address space.

However, the use of SPMs poses a number of new challenges. Indeed, from a software perspective the use and support of caches provides a good abstraction of a single shared memory space which simplifies programming since the programmer is freed from managing data movements and memory consistency. On the contrary, SPMs are visible to the programmer/software who has a disjoint view of the different levels of memories and is responsible of *explicitly* managing data movements by deciding what data to move, where and when. Machines with explicitly managed memory hierarchies will become increasingly prevalent in the future [KYM+07].

To improve performance, SPMs are usually combined with a hardware support for accelerating data transfers, called *Direct Memory Access* (DMA) engines. DMAs can transfer large amount of data between memory locations without processor interven-

tion offering another level of parallelism by overlapping computations and data prefetching [Gsc07, SBKD06] and thereby hiding memory latency. *Multi-buffering* programming schemes are often used to take advantage of this. We detail these aspects in the next chapter.

The idea of data prefetching to improve memory performance, in cache based architecture, it is a well and intensively studied topic. Several techniques have been proposed ranging from purely hardware prefetching solutions such as [DDS95, fClB95, Fri02] requiring a hardware unit connected to the cache to handle prefetching at runtime but at the cost of extra circuitry, to software prefetching approaches such as [MG91, CKP91] relying on compiler's analysis to insert additional fetch instructions in the code. Other works such as [CB94, WBM+03] combine both hardware and software prefetching approaches.

However, as mentioned previously, in the context of caches, data prefetching is transparent to the user and fetched data granularity is fixed to a cache line whereas in SPMs context this is not the case. In explicitly managed memory architectures, the effort of data movement is delegated to the programmer who has to make decisions about the granularity of data transfers and the way they are scheduled to achieve optimal performance. Indeed the programmer is solely responsible of setting up and sizing data buffers, managing alignment of data, synchronizing computations and data transfers and maintaining data coherence. Obviously this comes at the cost of programmers productivity and optimal performance can only be achieved if the programmer has a good understanding of the underlying architecture.

In order to improve both performance and programmers productivity we need to rely on adequate compiler and runtime support. Some new programming models such as Cellgen and Sequoia [FHK+06, SYN09, YRL+09] provide a programming front-end for explicitly managed memories. However, advanced compilers that are able to generate automatically efficient code based on optimal data movements decisions are still needed. This thesis suggests models that can aid programmers/compilers in optimizing such decisions.

DRAM

The reason why access to a cache/SPM is much faster than accessing main memory is that the physical structure of both is different. Main memories are usually built using Dynamic Random Access Memory (DRAM) technology whereas caches/SPMs using Static Random Access Memory (SRAM). The low price of DRAMs make them attractive for main memories despite being slower and more power hungry than SRAMs.

Unlike SRAMs that have a fixed access latency, DRAM latency is subject to variabilities. The main reason is that an SRAM memory is built of simple modules while a DRAM memory admits several complex features which eventually influence the latency. In the following we detail some of these features,

1. *Data refreshment*: SRAMs and DRAMs differ in the way they hold data. To store data, DRAMs use capacitors that leak power over time, therefore data needs to be *refreshed* periodically which means that information needs to be read and written again every few milliseconds, which distinguish the term dynamic in DRAMs from static in SRAMs. This refreshment induces an additional latency and requires extra circuitry which also adds to the system cost.

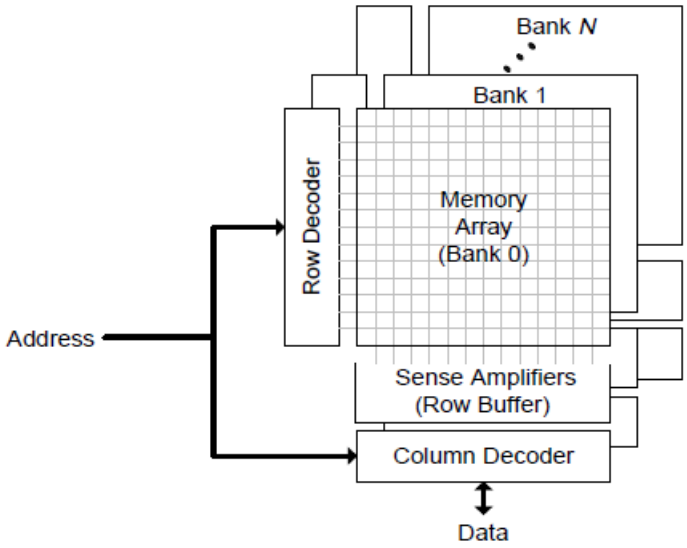


Figure 1.7: Modern DRAMs organization.

2. *Hierarchical organization*: modern DRAMs have a hierachical organization, illustrated in Figure 1.7, to store data. The core memory storage is divided into multiple banks where each bank consists of a rectangular matrix addressed by rows and columns such that a typical DRAM memory address is internally split into a row address and a column address. The row address selects a page from the core storage, and the column address selects an offset within the page to get to the desired word. When a row address is selected, the entire page addressed is *precharged* into the page buffer which acts like a cache making subsequent accesses to the same page very fast. In order to improve performance, each bank has an independent page buffer to increase parallel read/write requests to the DRAM since two separate memory pages can simultaneously be active in their page buffers. Therefore data layout in main memory plays an important role in performance where a contiguous access is less likely to be subject to page misses than a more fragmented access to memory.

3. *Requests scheduling*: DRAMs are usually coupled with a memory controller and a memory scheduler to arbitrate between concurrent requests to the DRAM, these requests are usually rescheduled to maximize memory performance by maximizing page hit rate. Therefore the arrival sequence of read/write memory requests also influences performance. This sequence includes requests issued from the multi-core fabric as well requests issued from other devices on the SoC, since access to main memory is shared between these devices. This can be an important source of variability even if to some extend we can have control on the read/write requests issued by the software running on the multi-core fabric.

In the first part of this chapter, we have set up the hardware architectural context of this thesis. In the sequel, we talk about some general issues concerning the software/application layers on the top of the hardware.

1.2 Embedded Software

Today the semiconductor industry has done great steps in building efficient multi-core platforms that offer several levels of parallelism. However, exploiting the provided hardware capabilities of modern multi-core architectures for the development of efficient software still remains the critical path to fully take advantage of this hardware [MB09].

Because of the increasing demand of integrating more functionalities in embedded architectures, embedded applications are becoming more computationally intensive, performance demanding and power hungry. In order to satisfy such constraints, programmers are required to deal with difficult tasks such as application/data partitioning, parallel programming and mapping to the hardware architecture, that govern performance. Below, we discuss some of these issues.

1.2.1 Parallelization potential of an application

For sequential programs, increase in the clock speed of the processor automatically increases the speedup of execution. For parallel programs, this is not necessarily true as we increase the number of processors since the benefit from parallel execution mainly depends on the inherent parallelization potential of the application. For instance, parallelizing video decoding standards such as MPEG and VC1 is limited by the parallelization of the Variable Length Decoder (VLD) kernel of code which is very challenging since the VLD algorithm requires a sequential access to the bitstream.

This idea is the essence of Amdahl's law [Amd67] stating that the benefit from parallelizing tasks can be severely limited by the non parallelizable (sequential) parts in a program. Let f be the fraction of the sequential part of a program, the parallel execution time given p processors is $f + (1 - f)/p$, since the non parallelizable part takes the same time f on both sequential and parallel machines and the remaining $(1 - f)$ is fully parallelizable. Therefore the speedup of the parallel execution defined as the ratio between the sequential and the parallel execution time is,

$$\text{Amdahl's law speedup} = 1/[f + (1 - f)/p] \ < 1/f$$

which is clearly bounded by the sequential part giving a maximal theoretical speedup of $1/f$. Figure 1.8 plots this speedup for different values of f as we increase the number of processors. Thus for a fraction of $f = 10\%$ ($(1 - f) = 90\%$), the speedup that can be achieved is at most $\times 10$ even if an infinite number of processors is provided. Hence, this law puts a limit on the usefulness of increasing the number of processors. Note that in practice this limitation is much more severe as Amdahl's law takes into account only the number of processors and ignores the effect of synchronization and communication.

Amdahl's law may seem as a strong limitation, however in practice, fortunately, the sequential overhead is very small for many applications. Furthermore, in many cases, the sequential part is constant or does not scale proportionately to the size of the problem as compared to the parallelizable part, making this limitation smaller as the size of the problem increases. *Embarassingly parallel applications* refer to the ideal case where the sequential part is null or negligible and therefore all computations in the program can be done in parallel.

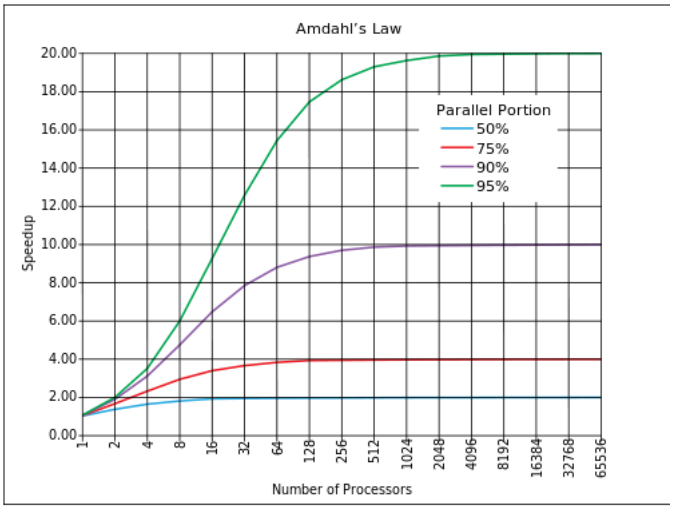


Figure 1.8: The limit on speedup according to Amdahl's law, the parallel portion is defined as $(1 - f)$.

1.2.2 Parallelization

Given an application, parallelization refers to the process of identifying and extracting parts of the application that can be executed concurrently. Basically, there are 3 forms of parallelism that we detail in the following,

Task parallelism refers to *different independent* activities that can run in parallel. These activities can be characterized by a task graph, a Directed Acyclic Graph (DAG) where nodes represent tasks and arrows precedence and data dependencies between them, see Figure 1.9. As for the granularity of tasks, a task may represent an instruction as well as a kernel of computation code or control code. Note that the sequential part of a task graph is defined by the critical path of the graph, it is the longest sequence of sequential tasks and it defines a lower bound on the execution time of the graph. The width of the graph corresponds to the maximum number of tasks that can run in parallel provided enough processors.

Data parallelism refers to different instances of the *same task* that are executed concurrently on *different data*. This form of parallelism can also be represented using a task graph, see Figure 1.9 where a fork and join tasks are added to ensure the synchronization between the beginning and the end of the task execution. Single Instruction Multiple Data (SIMD) and Single Program Multiple Data (SPMD) are very common forms of such parallelism. The main difference between them is the granularity of tasks and data where the former focuses on word level data combined with low level instruction tasks and the latter on coarse granularity data combined with program tasks. SIMD operations usually have a hardware support known as vector processors. Post decoding algorithms such as noise filtering that are used to improve the quality of decoded images are good candidates for data parallelization. Data parallelism is by far a better candidate for automatic extraction of parallelism, the parallelization of loops in compilers [DRV00] being a widespread example.

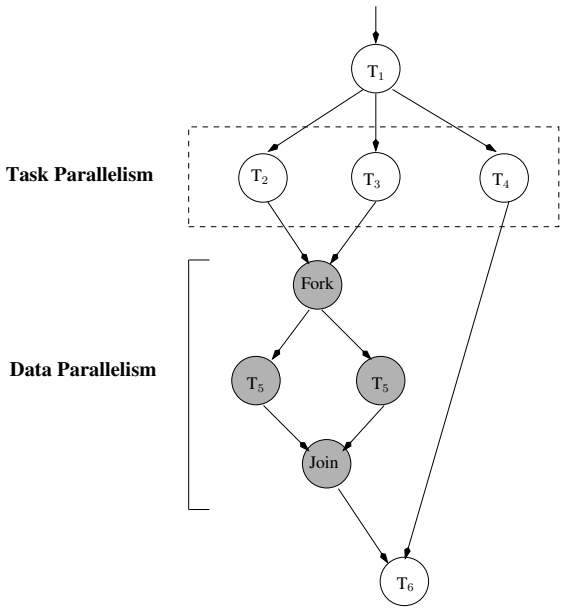


Figure 1.9: Task and data parallelism.

Pipeline parallelism refers to computations that form a sequence of stages where each stage corresponds to a different task/action performed by a different processor. Streaming applications [TKA02] such as video decoding algorithms performing a series of transformations on streams of data is a good example of pipelined parallelism. Figure 1.10 illustrates an 4-stages application F, computing a stream of data where each data x in indexed by its position i in the stream. For a given data item, there is a sequential execution of all stages so that the completion of one stage triggers the execution of the next one exhibiting a producer/consumer kind of interaction between successive stages. However, these stages can work concurrently on successive instances of the data stream. A N-stages pipelined application can have up to N tasks working in parallel at a given time. Dataflow paradigm [LM87] provides a natural way for modeling such applications that are viewed as a set of autonomous actors communicating through FIFO channels and where the arrival of data to a given stage triggers the execution, called firing, of this stage.

Luckily embedded applications exhibit a large degree of parallelism by combining different forms of parallelism and at different levels of granularity. However, this makes the task of automatic extraction of parallelism more difficult since it is critical/difficult to leverage the right combination of task, data and pipeline parallelism.

1.2.3 Parallel programming

After being restricted to some specific domains such as High Performance Computing (HPC), parallel programming is now becoming a main stream in software development because of the increasing use of multi-core platforms. However, writing efficient parallel programs has always been a very difficult task.

Several programming models and tools have been proposed in the recent years that aim at facilitating software development for MPSoCs. Commonly, these programming models are very low level libraries highly dependent on the target platform. They give the programmer full control over the architecture thus allowing him or her to provide a

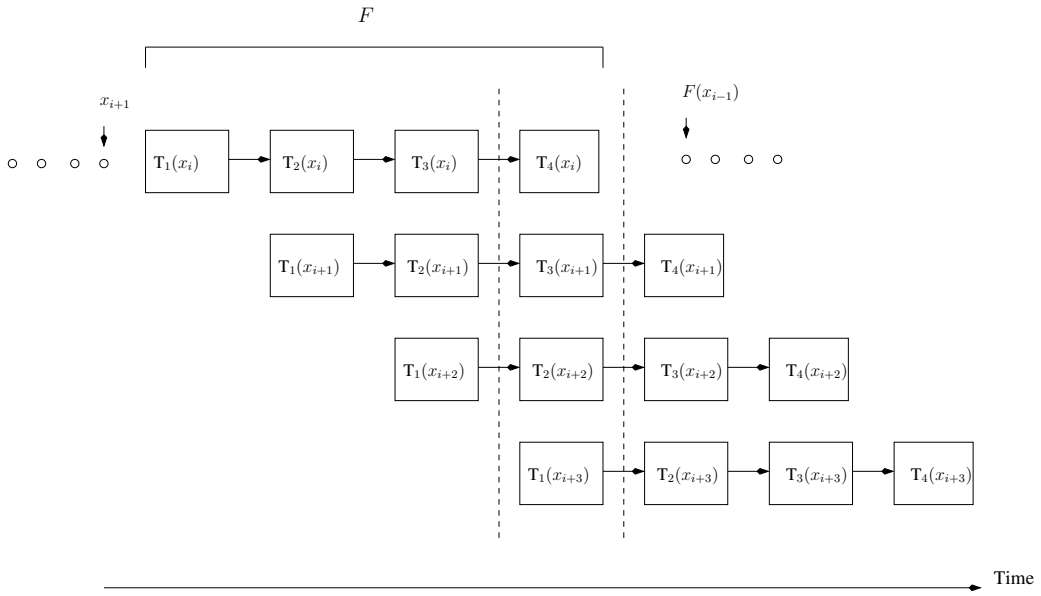


Figure 1.10: 4-stages pipeline execution of a stream of data.

fully optimized software. However, to achieve this, the programmer requires a very good understanding of all the platform low level hardware details which comes at the expense of a high development effort besides abstraction and code portability.

Therefore, today there is a need for providing a *standard* way for programming applications on multi-core architectures. Standard parallel programming frameworks, such as OpenMP [Ope08] initially designed for SMP architectures, are now used for programming MPSoCs. This requires a customized and effective implementation of the standard programming model constructs/directives on the underlying architecture and potentially extensions to match the target hardware. Works such as [MBB10] and [MB09] investigate and supports efficient implementation of OpenMP on MPSoCs featuring explicitly managed memory hierarchy.

One major issue to be aware of is that standard programming models solve the problem of functional/code portability but not performance portability. OpenCL [Gro08], a recent standard initially designed to be cross-platforms as it offers a unified framework for programming heterogeneous systems which can provide a mix of multi-core CPUs, GPUs, MPSoCs and other parallel processors such as DSPs, also poses the problem of performance portability. OpenCL is now capturing the interest of both academia and industry communities.

Another emerging concern in programming multi-core platforms is how to keep pace with cores *scalability*. Indeed, with the rapid increase in the number of cores, the limiting factor in performance will be the ability to write and rewrite applications to scale at a rate that keeps up with the rate of core count.

1.2.4 Deployment

The deployment step makes the link between the parallelized application and the underlying architecture. Deploying an application on a multiprocessor architecture is about deciding the *spatial* (mapping) and *temporal* (scheduling) allocation of resources to the

tasks.

The embedded feature of MPSoCs requires the applications implementation to meet real-time constraints under strict cost and power budgets. To achieve these requirements, both mapping and scheduling have to be done taking into account numerous criteria such as workload balancing, energy consumption and data communication. This is naturally formulated as constrained multi-criteria optimization problems [Ehr00] where decision variables corresponds to the allocation of tasks to the available resources, constraints define feasible solutions and the cost functions define the criteria to optimize expressed over the decision variables. When some of the criteria are conflicting, there is no single optimal solution for which all criteria are optimal but a set of incomparable efficient solutions known as Pareto solutions that represent the different trade-offs between conflicting criteria. Methods/tools such as [LGCM10, LCM11] can then be used to find or approximate the Pareto solutions.

To explore such solutions, the entry point of these tools is usually abstract models of both the application and the platform. The model of the application usually assumes a given parallelization that captures concurrency along with task and data dependencies and it is annotated with information relevant to the deployment decisions, such as the duration of tasks and the communication volume between two tasks. The model of the architecture exhibits the resources required for deployment such as the number of processors, the number of communication links between processors and the routing function, that are also annotated with information such as the frequency of processors, the network bandwidth. The choice of these parameters obviously depends on the criteria to optimize.

As to the granularity of the models, there is a clear trade-off between accuracy and efficiency, when more details are modeled, the gap in performance between the theoretical optimal solution and the practical solution is reduced, but this comes at the cost of more complexity and thus efficiency.

In a previous work [CMLS11], we explored the use of Satisfiability Modulo Theory (SMT) solvers [ZM02], to solve a simplified form of the mapping problem considering two conflicting criteria, that is computation workload and communication volume, on a distributed memory multi-processor architecture. This work is out of the scope of this thesis.

1.3 Conclusions

The correlation between hardware and software is very strong in MPSoCs, which are usually designed for a specific domain of applications and where optimal energy consumption and performance efficiency are achieved by hardware implementations at the expense of flexibility.

The shift of industry to more flexible architectures with more general purpose processing units puts a heavy burden on the software/application programmers who cannot ignore anymore the parallel features of the multi-core hardware and where optimal performance can only be achieved by a tuned and low level implementation of the application thus being very closely coupled with the target architecture. This comes at the expense of programmers productivity and software portability.

Therefore, in order to be efficient as well as flexible and portable, there is no escape from dealing with hard problems such as parallelizing applications, the error prone parallel execution of programs and deployment decisions considering numerous and con-

flicting criteria, to fully take advantage of the different levels of parallelism offered by the hardware as well as the memory hierarchy.

In this chapter, we presented a class of MPSoCs platforms that are the main focus of this thesis. Their main feature is an explicitly managed memory hierarchy where data movements are managed by the software using DMA engines, thus offering an opportunity for optimizing data transfers between the off-chip memory and the limited capacity on-chip memories, which is crucial for performance. This context changes the formulation of the classical parallel processing problems where no memory space constraints are assumed and where the main focus is to increase the processing capability while reducing the communication cost between processors, without any concern about data movement in the memory hierarchy.

Chapter 2

Preliminaries

2.1 Introduction

In this chapter, we first describe DMA engines, a hardware component which plays an important role in the performance of MPSoCs that need to transfer large data sets. We then define the structure of the class of applications on which we focus in this thesis, namely data parallel applications, and describe how these applications are programmed in MPSoCs in which data movements are performed by the software using *explicit DMA calls*.

2.2 Direct Memory Access (DMA) Engines

DMA is a hardware device that provides a *cheap* and *fast* way for transferring data. It is an important feature of many hardware systems including hard disk drive controllers in general purpose computers, graphics cards and systems-on-chip.

The main job of the DMA is to copy data (and sometimes code) from one memory location to another without involving the processor. When a processor needs data, it issues a command to the DMA by specifying the *source* address, the *destination* address and how much data it needs (number of words) and the DMA controller (DMAC) then takes charge of the transfer. When the data transfer terminates, the DMAC notifies the processor of its completion. DMA is more efficient for transferring a large quantities of data usually referred to as a *block*.

In MPSoCs, DMAs are particularily useful to relieve the cores from costly data transfers between off-chip memory and on-chip memory where a read or a write operation takes hundreds, sometimes thousands, of cycles. The implementation of data intensive applications, that constitute a large part of today's applications, impose a frequent access to the off-chip memory to transfer large data sets due to the on-chip memories limited capacity.

In order to understand the DMA behavior, we detail in the sequel the flow of a basic DMA transfer in the Cell B.E. architecture. We then present and discuss general DMA features, common to most architectures.

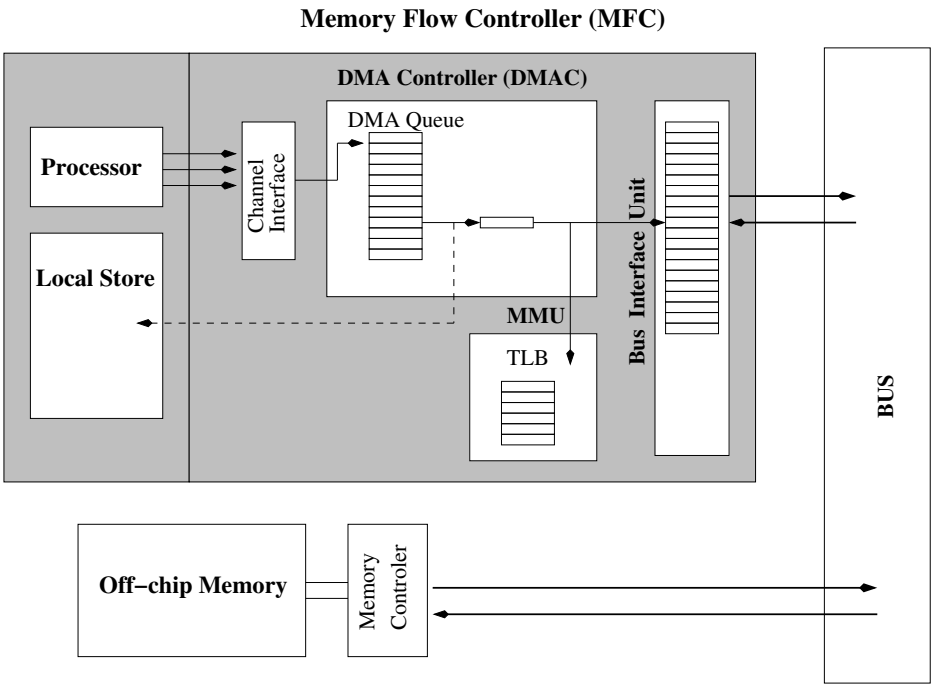


Figure 2.1: DMA command flow in the Cell B.E. platform.

2.2.1 Example of a DMA Command Flow

The Cell B.E. is a multi-core heterogeneous architecture consisting of a host processor and 8 cores acting as co-processors used for code acceleration. All elements in the Cell B.E. are connected through a high speed interconnect bus which is the main communication infrastructure of the platform. Each processor has a local store, a scratchpad memory of a small capacity (256 Kbytes), and a Memory Flow Controller (MFC) to manage DMA transfers. Local memory is the only memory directly accessible to the processor, the DMA is used to access data in other processors local store or main memory. More details about the architecture are given in Chapter 5, dedicated to the experiments.

Figure 2.1 presents a simplified overview of the components involved in a DMA transfer flow between a processor's local store and main memory. A more detailed description can be found in [KPP06]. When a processor needs data, it sends a transfer request to the DMA Controller (DMAC) through a channel interface by specifying all the parameters required for the transfer, and the command is then inserted in a DMA queue. In the Cell B.E. platform, the DMA queue has 16 entries and in case the queue is full, the processor is blocked from issuing other requests. Note that the Memory Flow Controller (MFC) has multiple channels enabling the DMA to receive multiple transfer requests.

The DMA controller (DMAC) selects from the queue a command to serve. Note that the order in which the commands are served is not necessarily the same order in which they arrive. The selected command is then submitted to the Memory Management Unit (MMU) which performs an address translation of source and destination addresses since the Cell B.E. uses a virtual memory addressing. The MMU uses a Translation Look-aside Buffer (TLB) for caching the results of recently performed translations.

After the address translation, the DMAC unrolls the DMA transfer command to create a sequence of smaller bus transfer requests since typically a DMA command granularity

is larger than the bus transfer granularity, in the Cell B.E. a bus request can transfer up to 128 bytes. These bus requests are stored in the Bus Interface Unit (BIU) queue. When the first bus request of a sequence of requests belonging to the same DMA command is selected from this queue, it is submitted to the bus controller and then to the off-chip controller and if it is accepted, data transfer begins to copy data from off-chip memory to the local store such that subsequent bus requests of the same command are pipelined. The DMA command remains in the DMA queue untill all its corresponding bus requests have completed. In the meanwhile, the DMAC can continue processing other DMA commands and when all bus requests of the current command have completed, it signals the command completion to the processor and removes the command from the queue.

In the Cell B.E. architecture, a DMA command can be a more complex object, that is, a *DMA list* to describe fragmented data transfers. Each list element is a contiguous transfer where the programmer needs to specify the source address and the destination address and the block size. These information are stored in the local store and when the command is selected from the queue to be processed, they are fetched to proceed for each list element, in the same way as previously, to address translation then splitting to several read/write bus requests.

2.2.2 The DMA's main features

So far, we presented a simplified view of the DMA behavior of the Cell B.E. . Some features vary from one architecture to another, however all DMAs share some common characteristics that we explain and discuss in this section.

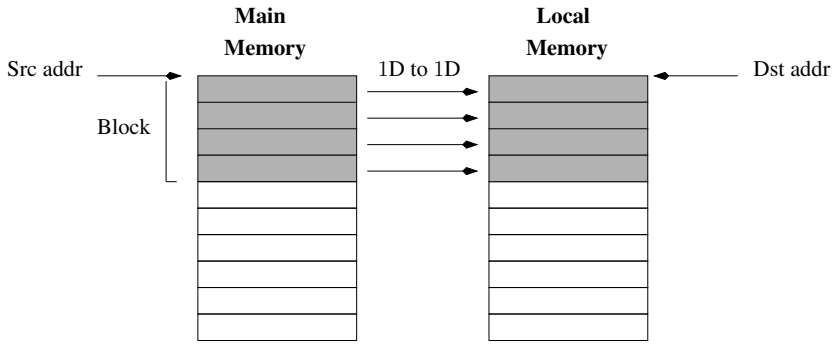
Overall, a DMA command flow can be decomposed into two major phases:

1. *Command initialization* phase: including the command issue time, the time to write the command in the queue and potentially some address translation when virtual memory addressing is used. Note that this phase is *independent* of the amount of data to transfer.
2. *Data transfer* phase: when a command is ready, data transfer begins, the block transfer request is then split into smaller read/write requests submitted to the interconnect and memory controller, these packets travel from source to destination through the on-chip/off-chip interconnect and then read/write to/from memory. The duration of this phase is clearly *proportional* to the amount of data.

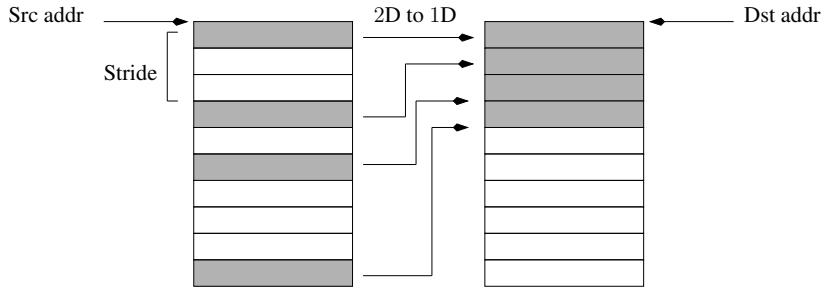
Note that each DMA request requires the allocation of a buffer in local memory, of the same size.

As mentioned previously, DMA engines are more efficient for *coarse granularity* data transfers than for low load/store instructions granularity typically because the initialization cost is significant and is only amortized for large data blocks. Furthermore, it can operate in *burst mode* (also called block transfer mode) where a block of possibly hundreds or thousands words/bytes can be transferred before the processor issuing the transfer is notified of its completion. Burst mode is very efficient since the memory latency is paid for the first word and then the remaining read/write requests of the same command are pipelined. This is very useful for loading program code and transferring large data sets required by data intensive applications.

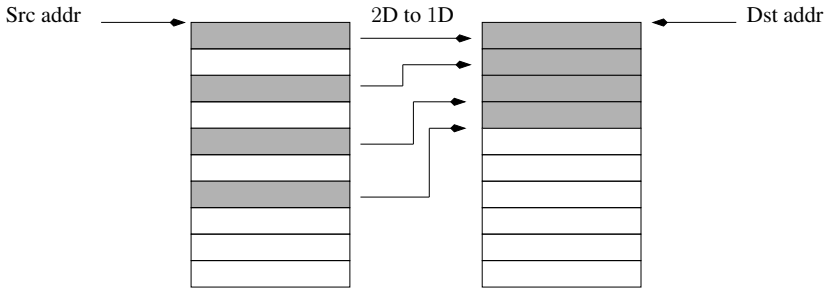
A DMA has multiple channels enabling it to receive several transfer requests and it has a scheduling policy to arbitrate between concurrent requests, sometimes based on programmable channels priority or on round robbin.



(a) Contiguous memory access.



(b) Strided memory access.



(c) Fixed stride memory access.

Figure 2.2: Different configurations for DMA data access.

DMA can transfer *contiguous* data blocks where data is stored in main memory as contiguous memory segments, as well as *strided* data blocks where data is fragmented in main memory. For such blocks, in addition to the source and destination address, a DMA command should specify a stride, an offset to access the next contiguous block in memory. Figure 2.2 illustrates different configurations for accessing data in main memory, contiguous blocks in (a) and fragmented blocks with a variable stride in (b) and a fixed stride in (c). These blocks of data are stored contiguously in local memory. Strided DMA transfers are in particular used for transferring rectangular data blocks required for applications working on two (or more) dimensional data arrays. In the Cell B.E. , there is no hardware support for strided DMA commands that are implemented at the software level using DMA lists which can be viewed as an array whose entries are pairs consisting of a main memory address and a contiguous transfer size.

In terms of performance, strided DMA transfers are costlier than contiguous DMA

transfers of the same size mainly for two reasons. The first concerns the initialization phase overhead which may be associated with each line of contiguous transfer if no hardware support for strided accesses is available. In the Cell B.E. , this corresponds to the software issue overhead for each list element and to the time to fetch information required by each list element transfer from the local store. Even with a hardware support, the issue overhead still remains more expensive because more parameters are involved in the command. The second reason concerns the physical matrix structure of DRAMs described in section 1.1.3. Since two dimensional data is usually organized in row major format in main memory, performing a stride to access next contiguous data in memory may require the precharging of a new page thereby inducing an additional latency overhead. Note that when the granularity of transfers is fixed (e.g a macroblock in video encoding) , it is possible to reorganize data layout in memory so that each unit of transfer forms a contiguous block in memory in order to reduce both the issue overhead and the page miss effect. However this processing overhead may also in practice hinder the performance.

DMA mechanisms for multi-core systems can be roughly classified as *centralized* (one device serves all processors) which is the case in the P2012 where one DMA is shared among processors in the same cluster, or *distributed* (each processor has its own DMA engine) like in the Cell B.E. .

The main limiting factor for DMA benefits is contention which has two sources,

1. The *on-chip network traffic*: resulting from simultaneous transfer requests of multiple processors sharing the same transfer infra-structure and more generally from the NoC traffic. Some of these contentions overhead can be controlled/reduced with an appropriate transfer scheduling policy. Having a centralized DMA system has the advantage of giving such control, especially for applications that exhibit a regular memory access pattern and move large amounts of data.
2. The *off-chip memory traffic*: the DRAM memory controller becomes the bottleneck for performance due to contentions resulting from both on-chip and off-chip concurrent read and write requests. This issue is very complex to handle (predict/manage) precisely as it requires an accurate model of the external DRAM characteristics such as the scheduling policy of the memory controller and the effect of page misses and data refreshment latencies. Furthermore, memory controllers are usually off-the-shelf IPs that vary among vendors and we usually have little control on them.

In MPSoCs, which are the main concern of this thesis, the strict memory and power constraints on one hand and the data intensive feature of the target applications on the other, render efficient use of DMAs essential for performance, conditioning to a large extent the success of MPSoCs. In such platforms, DMAs can be coupled either with caches or scratchpad memories. Unlike caches, where the prefetching is transparent to the programmer and the size of data to fetch is fixed to a cache line, in explicitly managed memories performance improvement can be achieved by an appropriate choice of DMA data granularity.

2.3 Data Parallel Applications

In this thesis we focus on data parallel applications for two major reasons, i) it is a common way for parallelizing code as they occur naturally and ii) they exhibit a regular

Algorithm 2.3.1 Sequential

for $i_1 := 0$ **to** $n-1$ **do**
  for $i_2 := 0$ **to** $n-1$ **do**
  $Y(i_1, i_2) := f(X(i_1, i_2))$
  od
od

memory access pattern and we can therefore reason about data transfers optimization statically.

Our typical structure of code is the computation $Y = f(X)$ described in the sequential algorithm 2.3.1 which *uniformly* applies f to the input array X to produce an output array Y where X and Y are *large* arrays of data [1]. For the sake of simplicity, we assume that both arrays X and Y have the *same* size and dimensions. In Program 2.3.1, X is a two dimensional data array of $n \times n$ elements, that we generalize thereafter to $n_1 \times n_2$ elements. We explore mainly this case as being the most interesting, the notation can then be generalized to any dimension.

In this section, we ignore the transfer from off-chip memory to the fabric as we assume that data is already available in the processors local memory, and we only focus on the logical structure of data. Therefore, we defer the discussion about the physical memory layout and the transfer/communication cost to the sequel.

2.3.1 Independent Data Computations

Given p processors, the input array can be then partitioned into p chunks of data processed concurrently, each processor computing n^2/p elements.

This is the simplest form of data parallelization, it can be viewed as a task graph illustrated in Figure 2.3 where a splitter and a merger task are added to insure the synchronization at the beginning and the end of the execution. Computations between processors are completely *independent* as there is no need for communication nor synchronization.

Note that there are two extreme cases in terms of sequential versus parallel execution, both are not realizable because of the following architectural limits,

1. A fully sequential execution requires the whole array to fit in the processor's local memory which, in the context of MPSoCs, is not possible because of the limited on-chip memories capacity that are typically smaller than n^2, this issue will be further discussed in the following sections.

2. A fully parallel execution requires enough processors so that each processor computes one array element. Typically, the number of available processors r in a multi-core fabric satisfies $r << n^2$.

Note that the model in Figure 2.3 only captures concurrency of data parallel tasks but tell us nothing about the structure of data or how it is allocated/mapped to each processor which both in practice have an important role when data communication and transfers are considered.

1. For instance, a full High Definition (HD) image consists of 1920x1080 pixels.

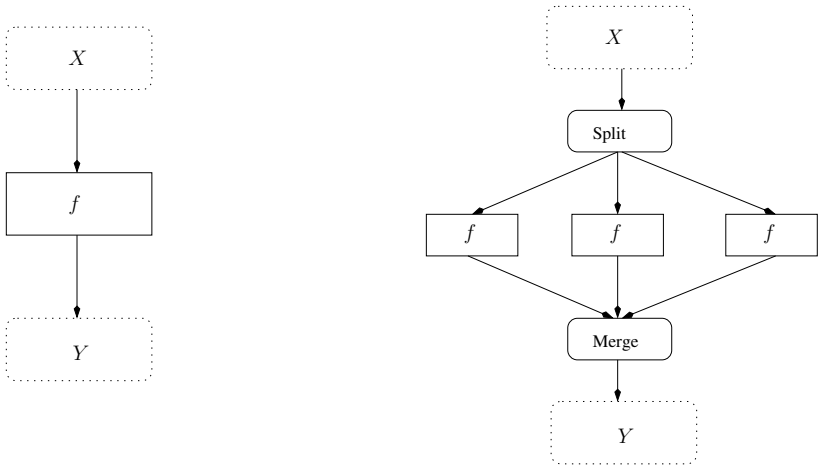


Figure 2.3: A fully data independent computation before and after data parallelization.

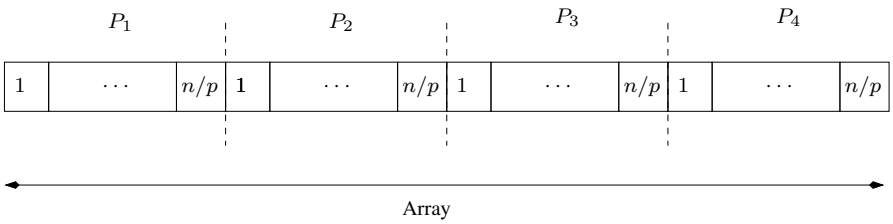
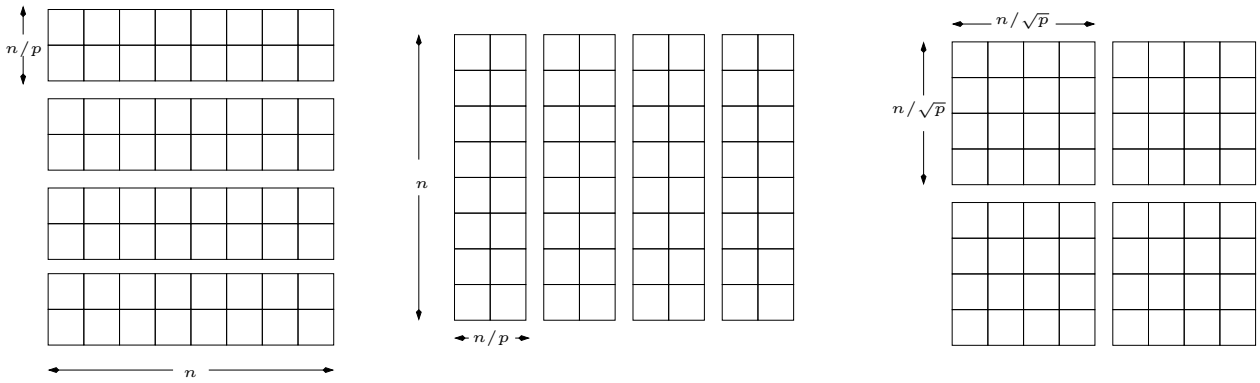


Figure 2.4: One-dimensional data partitioning.

Data Partitioning (size vs. shape) For one-dimensional data arrays, the choice of partitioning into p chunks is straightforward where each chunk is a contiguous block of n/p elements, as illustrated in Figure 2.4. However, for two-dimensional data structures, the geometry of data offers different options for array partitioning, as an example Figure 2.5 depicts different partitionings where each chunk clusters n^2/p elements of different shapes. Note that in terms of quantity of data, these solutions are equivalent, however when considering DMA transfers from main memory then, as argued previously in section 2.2, strided DMA commands are more expensive than contiguous commands of the same size, making the first choice (Figure 2.5(a)) optimal.



(a) n/p lines of n elements elements (b) n lines of n/p elements (c) $(n/\sqrt{p})$ lines of $(n/\sqrt{p})$

Figure 2.5: Different data chunks with the same size but different shapes.

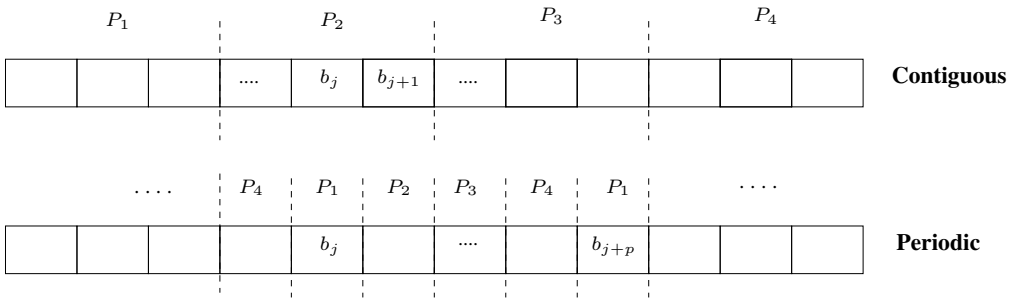


Figure 2.6: Contiguous vs periodic allocation of data blocks.

Data Allocation A chunk of data represents the total amount of work that each processor must perform. This chunk can be transferred to a processor's local memory at once or it can be further divided into smaller blocks, transferred to the local memory separately. This is particularly the case when the whole chunk of data cannot fit in the processor's local memory. Smaller blocks can then be allocated to the processors either in a *contiguous* manner where adjacent blocks j and $j+1$ are allocated to the same processor, or a *periodic* manner where blocks j and $j+p$ are allocated to the same processor, see Figure 2.6. Note that in terms of quantity of data these solutions are also equivalent, however as concurrent processors requests to access main memory are considered, periodic allocation of data blocks has the advantage of avoiding jumping from one memory location to another thus reducing page miss occurrence.

Because the reality of most algorithms today is more complicated than the simple form defined previously, we also focus in this thesis on a variant of these algorithms where computations share data. We refer to such computations as *overlapped* data applications.

2.3.2 Overlapped Data Computations

The main feature of these applications is that computations on each array element involve additional neighboring data. In image processing applications, these operations are for instance used in noise filtering algorithms and detection of local structures such as edges, corners, lines, etc.

For that, we assume a neighborhood function which associates, in a uniform manner,

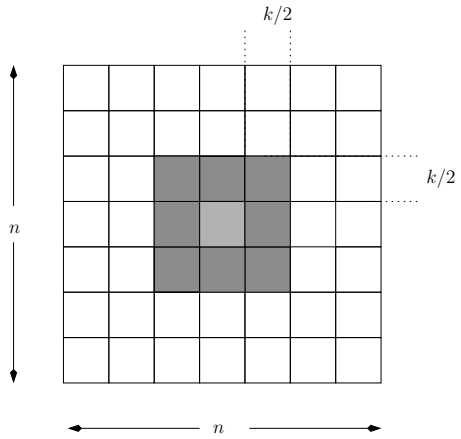


Figure 2.7: Neighborhood pattern of size k.

with every array element $X(i_1, i_2)$ a set of elements near to it including $X(i_1, i_2)$ itself. In other words, computation in the inner loop of Program 2.3.1 is replaced by,

$$Y(i_1, i_2) := f(V(i_1, i_2)),$$

where V is neighborhood data required for the computation. For instance,

$$V(i_1, i_2) = \{X(i_1 - 1, i_2), X(i_1, i_2 - 1), X(i_1, i_2), X(i_1 + 1, i_2), X(i_1, i_2 + 1)\}.$$

The Neighborhood dependency can be *spatial* when computations share *input* data, that is $V \subset X$, or *temporal* when $V \subset Y$. In this thesis, we only focus on spatial dependencies, temporal dependencies being more complicated since they create precedence between data parallel tasks which somehow combines data and task parallelism and requires more synchronization.

Both neighborhood pattern, size and the type of neighborhood dependency can differ from one application to another. This information is usually known a priori as it is part of the features of the algorithm.

As a neighborhood pattern, we consider a *symmetric* window of size k of input data as illustrated in Figure 2.7. Therefore, without loss of generality, we assume $V(i_1, i_2)$ to be a square around $X(i_1, i_2)$, that is,

$$V(i_1, i_2) = \left\{ X(j_1, j_2) : \begin{array}{l} (i_1 - k/2 \leq j_1 \leq i_1 + k/2 \\ (i_2 - k/2 \leq j_2 \leq i_2 + k/2 \end{array} \right\}$$

Note that such computations have a degree of spatial *locality*, since $k << n$.

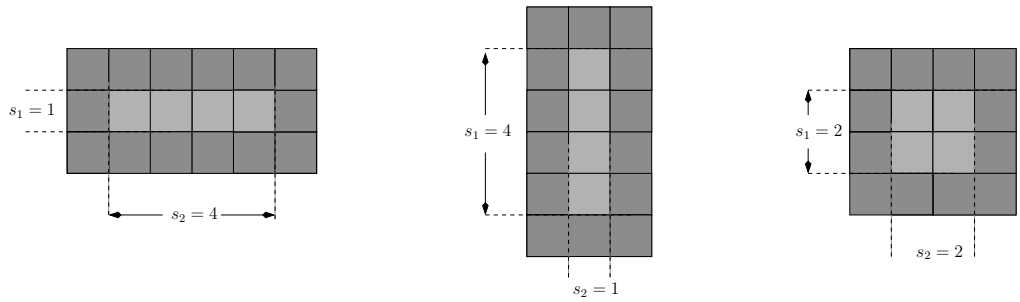


Figure 2.8: Influence of the block shape on the amount of shared data.

Data Sharing When data is partitioned among processors, the neighborhood pattern of V along with the geometry of data partitioning determine the amount of data *shared* between processors and consequently the synchronization and communication overhead that restrict the speedup obtained from the parallel execution. In one-dimensional data, the size of shared data is always fixed to k elements, no matter the size of the block. However, when considering two-dimensional data, the size and also the shape of the block influence the amount of shared data which constitute the *perimeter* around the block to compute. Suppose that the input array is partitioned into blocks clustering $s_1 \times s_2$ elements, The amount of shared data is therefore $k(s_1 + s_2) + k^2$. Figure 2.8 illustrates shared data for different block shapes of the same area $s_1 \times s_2 = \delta$. It is not hard to see that for each value of δ, shared data is optimized for square shapes, that is $(s_1, s_2) = (\sqrt{\delta}, \sqrt{\delta})$.

2.3.3 Discussion

While a lot of work has been done in the past to successfully parallelize data parallel applications in a multi-processor setting such as [AKN95, kLH95, AP01], contemporary MPSoCs architectures with a limited on-chip memory and a high latency access to main memory change the formulation and parameters of the problem and call for new solutions taking into account data layout in main memory along with the physical characteristics of both DMA and DRAM, in order to achieve high performance on these platforms.

For our work, we target a simple, yet a large enough class of applications that comprise a lot of today's applications, the most interesting features we considered are obviously the two-dimensional data structure and data sharing between neighboring computations. In the sequel we consider data transfers, and see how these simple data parallel loops defined in Program 2.3.1 are rewritten for such architecture to take into account DMA data transfers.

2.4 Software Pipelining

2.4.1 Buffering

In program 2.3.1 data arrays are initially stored in the off-chip memory. Therefore for an on-chip core to execute the program one needs first to bring the data from the external memory to a *closer* on-chip memory level typically using DMA. As mentioned

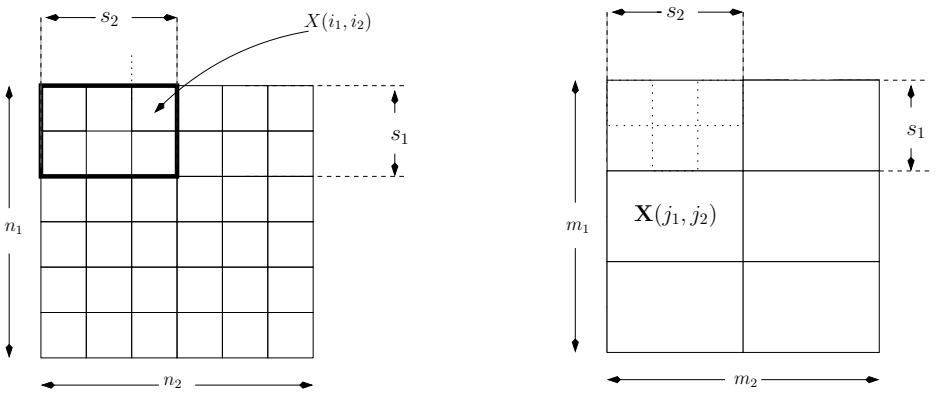


Figure 2.9: Basic blocks $X(i_1, i_2)$ and super blocks $\mathbf{X}(j_1, j_2)$ (logical view).

previously, transferring the whole input array to local memory, perform the computation then transfer the output array back to main memory is in practice not feasible because of the small size of the local memory, an SRAM of capacity tens/hundreds kilobytes.

We assume that an array element represents the *minimal granularity* for which the computation of f can be carried out. In image processing it can be a pixel, a block or a macroblock. We refer to such granularity as a *basic block*. An intuitive solution is to handle data transfers at the basic block level, but this is usually not a good choice for performance since DMA is more attractive for coarse data transfers.

Therefore, the input array is partitioned into *larger* data blocks that we assume rectangular (to keep the choice of the shape as general as possible) clustering $s_1 \times s_2$ basic blocks, as illustrated in Figure 2.9. We call such clusters *super blocks*, and they constitute the granularity of transfers. Note that the quantity $s_1 \times s_2$ is obviously constrained by the size of the local memory.

One can view the super blocks as arranged in an $m_1 \times m_2$ array $\mathbf{X}$ (and $\mathbf{Y}$) with $m_1 = n_1/s_1$ and $m_2 = n_2/s_2$. We use

$$\mathbf{X}(j_1, j_2) = \left\{ X(i_1, i_2) : \begin{array}{l} (j_1 - 1)s_1 + 1 \leq i_1 \leq j_1 s_1 \\ (j_2 - 1)s_2 + 1 \leq i_2 \leq j_2 s_2 \end{array} \right\}$$

to denote the set of basic blocks associated with a super block indexed by (j_1, j_2). It is sometimes more convenient to view two-dimensional arrays as one-dimensional and this is done by a flattening function $\phi : [1..m_1] \times [1..m_2] \rightarrow [1..m]$, for $m = m_1 m_2$. We will sometime refer to super block $\mathbf{X}(j_1, j_2)$ as $\mathbf{X}(j)$ for $j = \phi(j_1, j_2)$.

DMA transfers are explicitly inserted in the program code as shown in Program 2.4.1 where we use a *single* buffer B_x for input super blocks and another buffer B_y for output super blocks. Program 2.3.1 therefore becomes a sequence of computations and data transfers using *dma_get* and *dma_put* operations before and after the computation in the main program loop.

In program 2.4.1, data transfers and computations are performed *sequentially* and the processor is idle during reading and writing. To avoid limiting the performance potential of the processor, *asynchronous* DMA calls and double buffering are used.

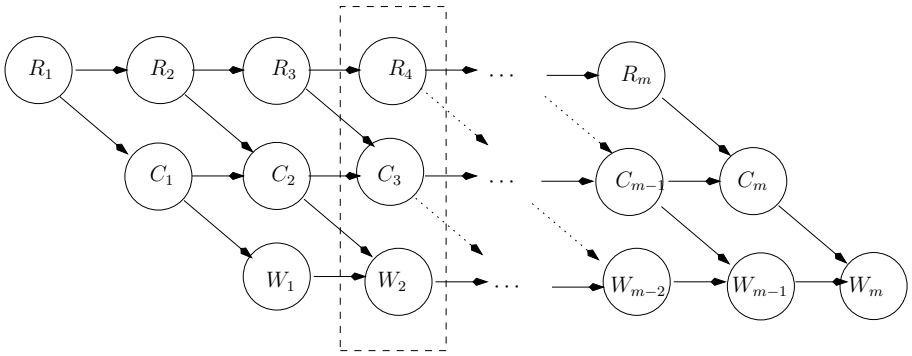


Figure 2.10: A schematic description of a pipeline: Read, Compute, Write.

2.4.2 Double Buffering

Double buffering or more generally multi-buffering is a well known programming technique referred to as *software pipelining* used to hide memory latency. The main idea is to overlap computations and data transfers and it is expressed at the software level.

This takes advantage from the fact that DMA can work in parallel with a processor. Program 2.4.2 uses *double buffering* for input blocks $B_x[0]$, $B_x[1]$ and output blocks $B_y[0]$, $B_y[1]$. At each iteration before the processor starts working on the current buffer, it issues an *asynchronous* DMA call to fetch the next buffer. Therefore, the processor can work on a super block j residing in one buffer while the DMA brings *in parallel* the super block $j + 1$ to the other buffer. Note that $comp(j)$ is used as a shorthand for the inner double loop in Program 2.4.1. In practice, events are associated with each *dma_get* and *dma_put* command to ensure the synchronization between each data transfer and its corresponding computation.

Hence, program 2.4.2 defines a *software pipeline* with 3 stages: input of super block $(j + 1)$, computation on super block j and output of super block $(j - 1)$, see Figure 2.10. Reading the first block and writing back the last block are, respectively, the *prologue* and *epilogue* of the pipeline. Note that Figure 2.10 only describes the obvious precedences between the computations and data transfers but tells us nothing about their relative durations.

Double buffering overlaps at each iteration only one input/output DMA transfer with the computation of current buffer. It is possible to increase the number of DMA transfers that are overlapped with one computation by using *Multi-Buffering*. Indeed when k-buffering is used, computation of super block j is done in parallel with transfers of input super blocks $(j+1), (j+2), ..., (j+k-1)$ and output super blocks $(j-k), (j-k+1), ..., (j-1)$. This is possible since DMA has multiple channels to store a sequence of transfer requests.

In terms of performance, since a DMA command has two phases, an initialization phase and a transfer phase as explained in section 2.2, then issuing a sequence of transfer requests has the advantage of overlapping in time the transfer of the current request along with the initialization phase of pending transfer requests. However, let us note the following facts:

1. Multi-buffering comes at the cost of a higher memory budget requirement. Indeed, excluding program code and additional working memory, when using k-buffering local memory should be large enough to store $2k$ local buffers, of $s_1 \times s_2$ basic blocks each. Therefore given a local memory budget, increasing the number of

buffers used will necessarily decrease the upper bound on the size of each local buffer knowing that if we decrease the buffer size too much, DMA use may become less efficient.

2. There is a limit of how much performance benefit we can withdraw from overlapping transfer phase of the current DMA request and initialization phase of subsequent requests. The gain is negligible if the initialization cost is much smaller than transfer cost, or on the contrary if the initialization time is significant compared to the transfer time and therefore performance becomes closer to a sequential execution of all transfers.

3. DMA requests are not necessarily served in the same order in which they arrive. Therefore if super block $j + 2$ is served before super block $j + 1$ then computation of $j + 1$ should wait for the completion of both transfers. One solution for this is the use of fences or barriers between transfer requests which comes with an additional cost overhead.

For these reasons we focus in this thesis on double buffering.

One can see how the simple parallel loop in Program 2.3.1 has been transformed to a more complex algorithm where the programmer has to manage multiple buffers, dealing potentially with border cases using if conditions when the size of the array is not a perfect multiple of the block size, and ensuring the synchronization between DMA fetching and computations, which complicates the source code and counts for a substantial number of lines in it. This exhibits some of the complex aspects of parallel programming for explicitly managed memories which can be hidden from the programmer when an efficient compiler is used for generating such code automatically.

2.5 Choosing a Granularity of Transfers

DMA combined with scratchpad memories give the programmer the freedom to chose the granularity of transfers, that is the block size and also shape for multi-dimensional data structures. Double buffering scheme improves performance compared to single buffering by interleaving computations and data transfers. However, performance can be further improved (controlled/tuned) by an appropriate choice of data granularity. A natural question that arises then is how to make this choice given the available local memory budget? the answer may not be straightforward since as we saw throughout this chapter, DMA performance as well as computations and the amount of shared data are sensitive to the size and the geometry of the block, thus the choice of data granularity influences performance and sometimes in a significant way.

We propose a general methodology based on hardware and application parameters to derive automatically the choice of data granularity that yields optimal performance within the available local memory budget. An obvious choice for programmers would be the maximum buffer size allowed by the local memory. However we show in this thesis that this is not necessarily the optimal choice.

This methodology comes in contrast with the engineering way for tackling this problem which consists of writing a double buffering algorithm, for a given application and a given architecture, in a parametric manner and run the code with different granularities and then select the best among them. Furthermore, our methodology is done statically

and can be combined with efficient compilers to generate automatically double buffering programs with the appropriate data partitioning and granularity.

Algorithm 2.4.1 Buffering

for $j := 1$ **to** m **do**
$dma\_get(B_x, \mathbf{X}(j))$; % read super block
for $i_1 := 1$ **to** s_1 **do**
  for $i_2 := 1$ **to** s_2 **do** % compute for all blocks
    $B_y(i_1, i_2) := f(B_x(i_1, i_2))$ % in super block j
  od
od
$dma\_put(B_y, \mathbf{Y}(j))$; % write super block
od

Algorithm 2.4.2 Double Buffering

$c := 0; c' := 1;$
$dma\_get(B_x(0), \mathbf{X}(1))$; % first read
$dma\_get(B_x(1), \mathbf{X}(2)) \parallel comp(1)$;
for $j := 2$ **to** $m - 1$ **do**
  $dma\_get(B_x(c), \mathbf{X}(j+1)) \parallel comp(j) \parallel dma\_put(B_y(c'), \mathbf{Y}(j-1))$;
  $c := c \oplus 1; c' := c' \oplus 1;$
od
$comp(m) \parallel dma\_put(B_y(0), \mathbf{Y}(m-1))$;
$dma\_put(B_y(1), \mathbf{Y}(m))$; % last write

Chapter 3

Optimal Granularity for Data Transfers

3.1 Computations and Data Transfers Characterization

To derive optimal granularity, we need first to analyze the performance of the pipelined execution of double buffering algorithms and understand how performance is influenced by the size and the shape of a super block, which constitutes the granularity of transfers. To this end we need to refine the qualitative description of Figure 2.10 which describes the obvious precedences between the computations and data transfers but tells us nothing about their relative durations. In the sequel, we characterize the DMA transfer time and the computation time of a super block, that we denote T and C respectively.

3.1.1 DMA Performance Model

It is quite difficult to model the DMA behavior precisely taking into account all low level hardware details that vary from one architecture to another. Nevertheless, all DMAs share some common characteristics as explained in section 2.2.2. In the following, we provide a simplified analytical DMA model that captures the important and common features of DMA engines. We specify the cost of transferring a contiguous cluster of basic blocks which, we recall, corresponds to the atomic granularity of an algorithm, we then extend this model to specify the cost of transferring a rectangular cluster of basic blocks using strided DMA commands.

For this, we need to make the following assumptions about data layout in main memory. We assume the array is organized in main memory contiguously in a lexicographic order (that is a row major format for two-dimensional data arrays). Furthermore, without loss of generality we assume that a basic (input and output) block consists of a contiguous chunk of b bytes. This model can easily be adapted to the case where the basic block is a rectangular block of b_1 lines, each line consists of b_2 bytes.

Contiguous DMA Transfers

As explained in section 2.2.2, a DMA command flow consists of an initialization phase that we assume of a fixed cost I and a transfer phase whose duration is proportional to the size of data to transfer. So given a transfer cost of α time units per byte, the transfer time of a super block clustering s basic blocks as illustrated in Figure 3.1, is then approximated by,

$$T(s) = I + \alpha \cdot b \cdot s.$$

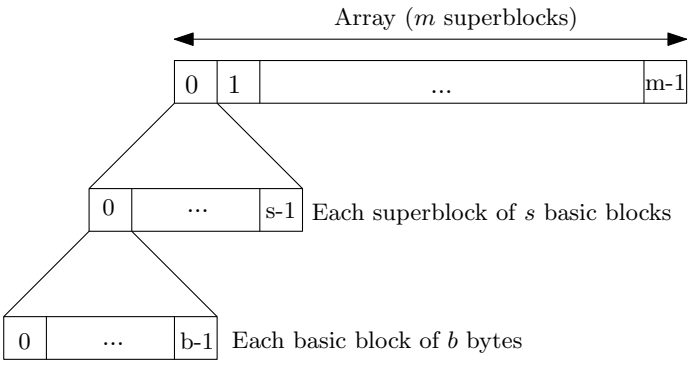


Figure 3.1: Decomposition of one-dimensional input (resp. output) array.

Strided DMA Transfers

As mentioned previously, to transfer rectangular blocks, strided DMA commands are used. The cost of transferring a super block of $s_1 \times s_2$ basic blocks, which corresponds physically to a rectangular data chunk of s_1 lines and $b \cdot s_2$ columns, can be approximated by the following affine function,

$$T(s_1, s_2) = I_0 + I_1 \cdot s_1 + \alpha\, b(s_1 \cdot s_2) \tag{3.1}$$

Like contiguous transfers, this function assumes a fixed initialization cost I_0 (typically $I_0 \geq I$) and a transfer phase whose duration is also proportional to the amount of data to transfer which corresponds in this case to the area of the super block. Furthermore, we assume an overhead I_1 per line to capture the fact that transferring a rectangular block is costlier than transferring a contiguous block of the same size (area), as explained in section 2.2.2. This fact is expressed by the following inequality,

$$T(s_1 \cdot s_2\ ,\ 1) \leq T(s_1, s_2)$$

Note that despite strided DMA commands being costlier than contiguous, they remain more efficient than using a separate contiguous DMA command for each line of data, a fact expressed by,

$$T(s_1, s_2) \leq s_1 \times T(s_2)$$

The value of α depends on several hardware factors such as interconnect bandwidth, memory latency (which is different according to the type of memory: SRAM of another processor or off-chip DRAM), and possible contentions. In this model we assume a *fixed* transfer latency α and this assumption is imprecise for two major reasons:

- We do not model the characteristics of the external DRAM memory explained in section 1.1.3, such as the scheduling policy of the memory controller, the effect of page misses and data refreshment latencies.
- The speed of transfer in the interconnect, especially in a multi-processor setting, depends crucially on the number of simultaneous transfer requests issued by the processors involved in the computation.

The first issue is too complex to handle precisely as memory controllers vary among vendors. We can assume, however, that page misses are distributed more or less evenly and their effect does not favor or disfavor a specific choice of granularity. Moreover, the

preference to contiguous blocks captured by our cost model by the parameter I_1 holds also on the memory controller side.

As for the influence of demand patterns on the latency of the interconnect, we will use later a model where α is parametrized by the number p of active processors with $\alpha_p < \alpha_{p'}$ whenever $p < p'$. Additional traffic, which possibly influence the value of α, resulting from other applications is ignored as we assume that there is only one application running on the multi-core fabric, which gives us full control on the generated NoC traffic.

3.1.2 Computation Time

Regarding computation time per super block, for the sake of simplicity, we assume the algorithm for computing f to have a fixed (data independent) computation time ω per basic block, once the block is in local memory. This is the time to perform one iteration in Program 2.3.1. This may sound as a strong assumption, however if the deviation in the computation time per basic block is not significant then we can assume that an optimal analytical choice of granularity based on an average value of ω still gives good performance results.

Computation time of a contiguous cluster of s basic blocks is therefore $C(s) = \omega \cdot s$ and for rectangular clusters, computation then depends only on the area of the rectangle,

$$C(s_1, s_2) = \omega \cdot s_1 \cdot s_2 \tag{3.2}$$

In practice, $C(s_1, s_2)$ has also a component that depends on the number of lines s_1 and which corresponds to the overhead at each computation iteration related to the setting required between the outer loop and the inner loop like adjustment of the pointers for every row, pre-calculation of sums of borders, etc. We assume so far that this overhead is negligible and we discuss it further in the chapter dedicated to the experiments.

3.2 Problem Formulation

As explained in the previous chapter, the execution of a double buffering program forms a 3 stages pipeline which admits repetitive parallel execution of computations (for super block i) and data-transfers (for input super block $i + 1$ and output super block $i - 1$).

Based on the balance at each iteration between the computation time of the current super block and the transfer time of the next block, we refer to the execution of the pipeline as being in the *computation regime* when the computation time of a super block dominates the transfer of the next one , that is $C > T$, otherwise it is in the *transfer regime*. The behavior of the software pipeline in both regimes is illustrated in Figures 3.2.

Assuming that the input and output super blocks are identical and can be transferred in parallel [1], it is not hard to see that both regimes admit a prologue and epilogue, which correspond to the transfer of first block and write back of last block and m episodes, m being the number of super blocks, dominated either by transfer or by computation. Therefore the pipeline execution time τ can be approximated by the following,

1. In practice, this is not completely true since input and output DMA transfers are overlapped, but we can assume that this simply varies the value of α.

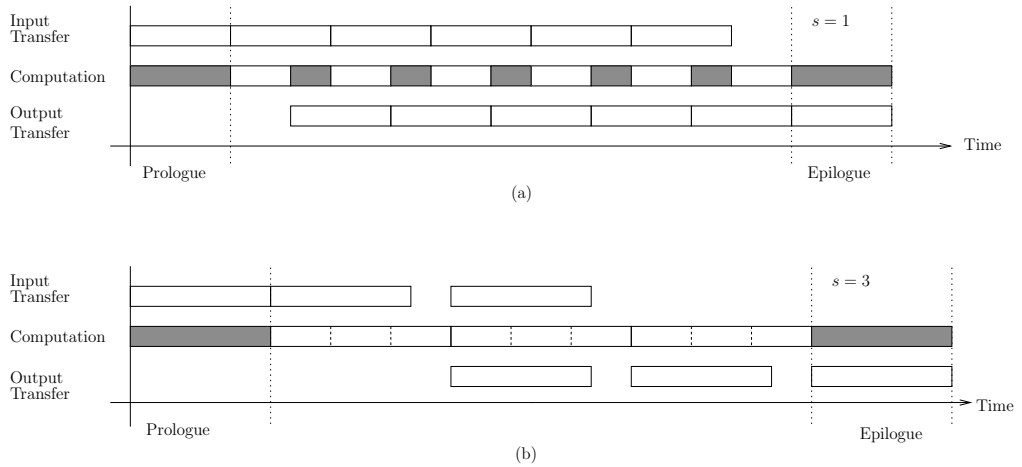


Figure 3.2: Pipelined execution using double buffering on one processor: (a) transfer regime, (b) computation regime. The shaded areas indicate processor idle time.

$$\tau = \begin{cases} m \cdot C + 2T & \text{in the } \textit{computation} \text{ regime} \\ (m+1) \cdot T & \text{in the } \textit{transfer} \text{ regime} \end{cases} \tag{3.3}$$

The ratio between computation time of a super block C and its transfer time T is not fixed but varies with the block size and shape. We can therefore control it to some extent in order to optimize performance, but which relation is preferred? The answer depends on which resource is more stressed by the application, computation or communication, a fact characterized by the parameter ψ so that,

$$\psi = \omega - \alpha b.$$

Condition $\psi < 0$ means that regardless of the choice of data granularity, transfer time always dominates computation time. In this case we prefer large data blocks (which corresponds to the maximal buffer size allowed by the local store capacity) to amortize the DMA initialization time and fully utilize the interconnect bandwidth. In the sequel, we focus on the other case where $\psi \geq 0$, that is according to the block size and shape the execution switches between a computation regime and a transfer regime.

For the same instance of the problem, computation regime yields better performance than transfer regime because the processor does not stall between two iterations waiting for data thereby avoiding idle time. Therefore we orient the super block selection towards a granularity (s for contiguous blocks and (s_1, s_2) for rectangular blocks) such that $C \geq T$ and $\tau = m \cdot C + 2T$ is minimal.

Since the processor is always busy, all shapes satisfying $C \geq T$ admit roughly the same total computation time $m \cdot C \backsimeq \omega n$ (or $\omega n_1 n_2$) since it is a sequential execution over all the basic blocks. Hence, it remains to optimize the length of the prologue and epilogue $2T$ with computation regime viewed as a constraint. Note that when ω or n are very large, the prologue and epilogue represent a small part of the overall performance and their variation has a negligible effect which in practice is not always the case. Obvious additional constraints state that a super block is somewhere between a basic block and the full image, provided its size does not exceed the maximum local buffer size M imposed by the local store limited capacity. This leads to the following constrained optimization problems,

$$\begin{array}{ll} \min\ T(s)\ \text{ s.t.} & \min\ T(s_1,s_2)\ \text{ s.t.} \\ \\ T(s) \le C(s) & T(s_1,s_2) \le C(s_1,s_2) \\ s \in [1..n] & (s_1,s_2) \in [1..n_1] \times [1..n_2] \\ b \cdot s \le M & b \cdot s_1 \cdot s_2 \le M \end{array} \tag{3.4}$$

Note that in addition to these constraints, each specific DMA engine imposes additional constraints on the range of possible values of s_1 and s_2.

In the following we derive the optimal granularity of data transfers starting with a single processor and then considering multiple processors.

3.3 Optimal Granularity for Independent Computations

3.3.1 Single Processor

One-dimensional data:

The ratio between computation time of a super block and its DMA transfer time splits the domain of feasible solutions into two sub-domains, the *computation domain* where each granularity choice guarantees a computation regime since its computation time is larger than its transfer time and the *transfer domain*.

Figure 3.3-(a) illustrates the intersection of functions $T(s)$ and $C(s)$ for one-dimensional data blocks where the computation domain corresponds to the interval $[s^*, n]$, and the overall execution switches from a transfer regime to a computation regime for granularity s^*, as illustrated in Figure 3.3-(b). It is approximated by

$$\tau(s) = \begin{cases} (n/s+1)\,T(s) & \backsimeq (n \cdot I)/s + (n \cdot \alpha \cdot b) \quad \text{for} \quad s < s^* \\ 2 \cdot T(s) + n \cdot \omega & \backsimeq (\alpha \cdot b)s + (n \cdot \omega + I) \quad \text{for} \quad s > s^* \end{cases}$$

As the granularity increases in the interval $[s^*, n]$, DMA transfer time also increases. Hence optimal granularity minimizing the prologue and epilogue in the computation regime is attained at s^* where $T(s^*) = C(s^*)$,

$$s^* = I/(\omega - \alpha b) \tag{3.5}$$

Note that if for any granularity s the execution is always in the computation regime, the optimal unit of transfer is a basic block, that is $s^* = 1$, which guarantees minimal prologue and epilogue.

Two-dimensional data:

For rectangular blocks, the dependence of $T(s_1, s_2)$ and $C(s_1, s_2)$ on their arguments is illustrated in Fig. 3.4 (assuming $\psi > I_1$). Similarly, the intersection of these two surfaces separates the domain of (s_1, s_2) into two sub-domains, the *computation domain* where $T \le C$ and the *transfer domain* where $T \le C$, see Fig. 3.5-(a).

As for one-dimensional data, we want to find the optimal granularity defined as the point (shape) in the computation domain for which transfer time is minimal. Comparing the transfer time of the different shapes becomes not trivial since the computation domain

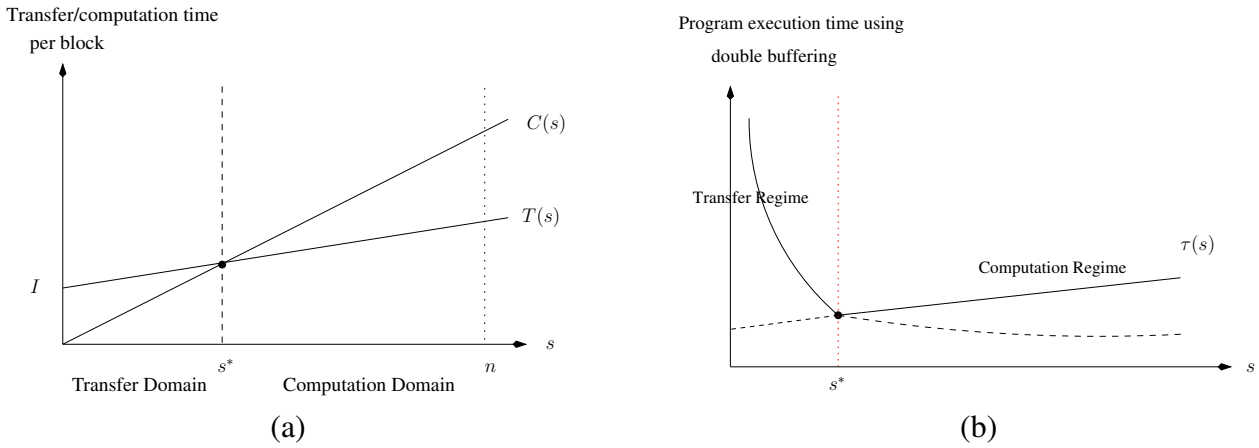


Figure 3.3: Contiguous blocks: (a) The dependence of C and T per block, (b) Pipeline execution time.

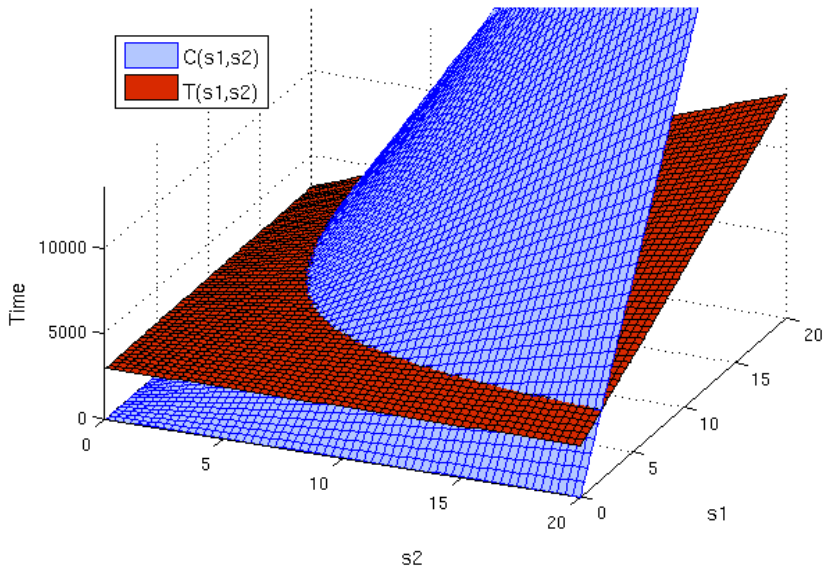


Figure 3.4: Rectangular blocks: The dependence of computation C and transfer T on the granularity (s_1, s_2).

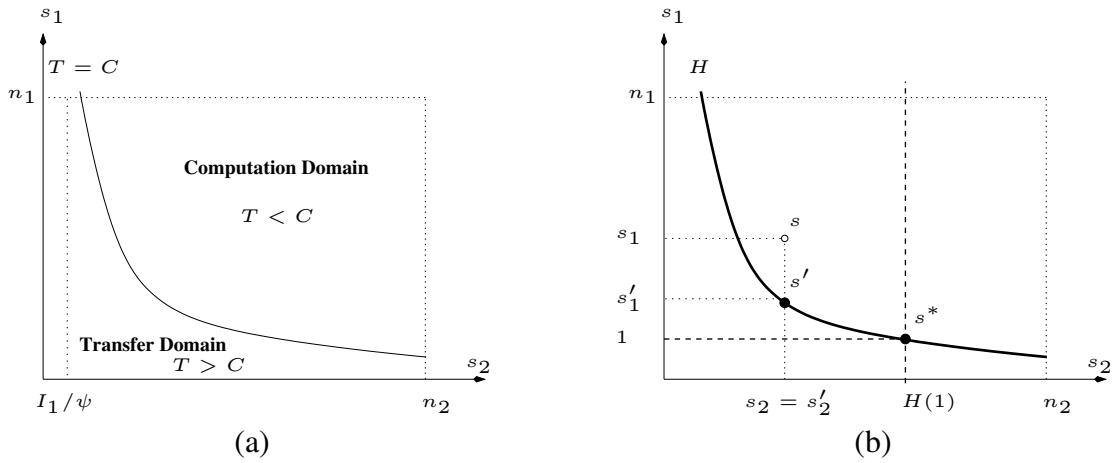


Figure 3.5: Rectangular blocks: (a) computation and transfer domains, (b) optimal granularity candidates and optimal granularity.

forms a partially ordered set where possibly two different shapes s and s' are such that $s_1 < s'_1$ and $s_2 > s'_2$ [2].

Observe that the computation domain is convex where for any point s inside the domain, we can always find another point s' on the boundary such that $s'_2 = s_2$ and $s'_1 < s_1$, see Fig. 3.5-(b), and hence with a smaller transfer time. Therefore the candidates for optimality are restricted, as for the one-dimensional case, to the *intersection* $T(s_1, s_2) = C(s_1, s_2)$. These points are of the form $(s_1, H(s_1))$ where,

$$H(s_1) = (1/\psi)(I_1 + I_0/s_1)$$

Their transfer time is expressed as a function of the number of clustered horizontal blocks s_1:

$$T(s_1, H(s_1)) = c\,(I_0 + I_1 s_1)$$

where c is the constant $1 + (\alpha b/\psi)$. This function is linear and monotone in s_1, means that as we move upwards in the hyperbola H the transfer time increases, and hence optimal shape is,

$$(s_1^*, s_2^*) = (s_1^*, H(s_1^*)) = (1, H(1)) \tag{3.6}$$

which constitutes a contiguous block of one line of the physical data array. This is not surprising as the asymmetry between dimensions in memory access prefers "flat" super blocks with $s_1 = 1$. Without data sharing and memory size constraints the problem becomes similar to the one-dimensional case where it is only the *size* of the super block that needs to be optimized.

3.3.2 Multiple Processors

Given p *identical* processors having the same processing speed and the same local store capacity, the input array is partitioned into p chunks of data distributed among the

2. Note that if these shapes have the same area, then the shape with less lines has a smaller transfer overhead, that is $T(s_1, s_2) < T(s'_1, s'_2)$.

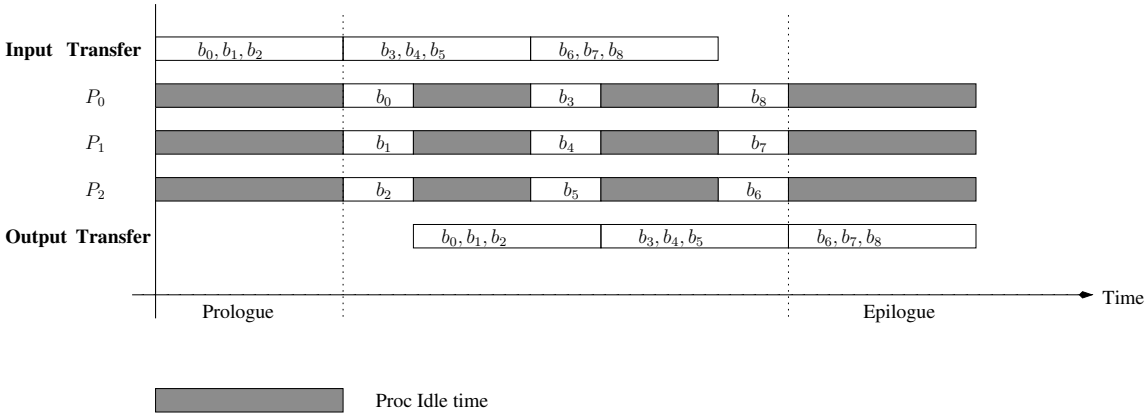


Figure 3.6: Pipelined execution in the transfer regime using multiple processors.

processors to execute in parallel. Typically the size of a chunk allocated to each processor is much larger than the local memory capacity since $p << n$ (or $p << n_1 n_2$), each processor then implements double buffering algorithm to improve performance by overlapping computations and data transfers. We extend our double-buffer granularity analysis to this case assuming all processors implement the same granularity.

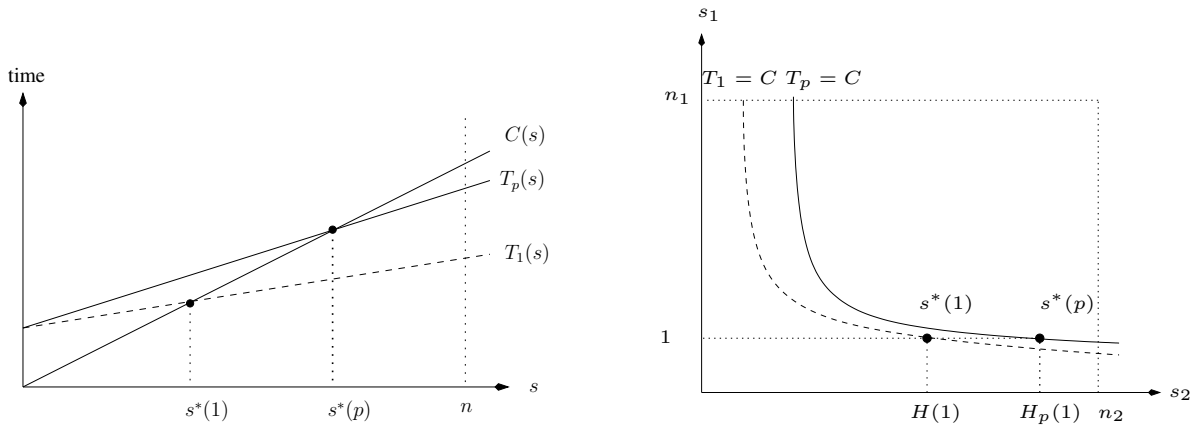
Intuitively, using multiple processors, a conflict arises between computation and data transfers since increasing the number of processors reduces the amount of total work per processor but creates contentions on the shared resources thus increasing the transfer time.

In the analysis, we assume a distributed DMA system where each processor has its own DMA engine. It has the advantage of parallelizing the initialization phase of processors transfer commands which occurs independently on each processor's DMA engine. For this reason we synchronize data transfers of all processors at the beginning of the execution [3]. Figure 3.6 illustrates a pipelined execution using several processors where p concurrent transfer requests arrive *simultaneously* to the shared interconnect. Arbitration of these requests is left to the hardware which serves the processors in a low granularity (packet based) round robin fashion. Therefore processors receive their super blocks nearly at the same time and can then perform their computations independently in parallel.

Note that increasing the number of processors does not influence the computation time per super block, however it increases the transfer time because contentions on the shared resources induce a significant overhead that we model by parameterizing the transfer cost per byte α with the number of active processors such that α_p increases monotonically with p.

Obviously this changes the ratio between the computation time and the transfer time of a super block and consequently the optimal granularity. Figure 3.7 shows the evolution of the computation domains and optimal granularity for one-dimensional and two-dimensional data as we increase the number of processors. The reasoning is similar to previously where functions T and H become T_p and H_p thus yielding an optimal data granularity for each value of p. Note that the difference between computation and transfer time represented by $\psi = \omega - b\alpha_p$ decreases as we increase p reducing the computation domain. Also, beyond some value of p we are always in a transfer regime and there is no

3. Note that the gain from overlapping the initialization phase is less significant for large granularities where the transfer phase time dominates the fixed initialization overhead.



(a) One-dimensional data (b) Two-dimensional data

Figure 3.7: Evolution of the computation domain and optimal granularity as we increase the number of processors.

point in using more processors since our main focus is to optimize processors idle time.

The overall pipeline execution time becomes also parametrized by the number of processors p as follows,

$$\begin{array}{ll} m/p \cdot C + 2T_p & \text{when } C \geq T_p \\ (m/p + 1) \cdot T_p & \text{when } C < T_p \end{array}$$

Optimal granularity, which is simply the necessary amount of data needed to hide memory latency, increases as we increase the number of processors since more data needs to be brought to each processor to keep it busy during the time it takes to fetch its next super block *as well as* the next super block of each of the other processors.

Note that with data independent computations, we do not distinguish between a cyclic and a contiguous allocation of data, explained in the previous chapter, as this simply affects the value of α which maybe larger because of the page miss effect as we jump from one memory location to another as argued previously.

3.3.3 Memory Limitation

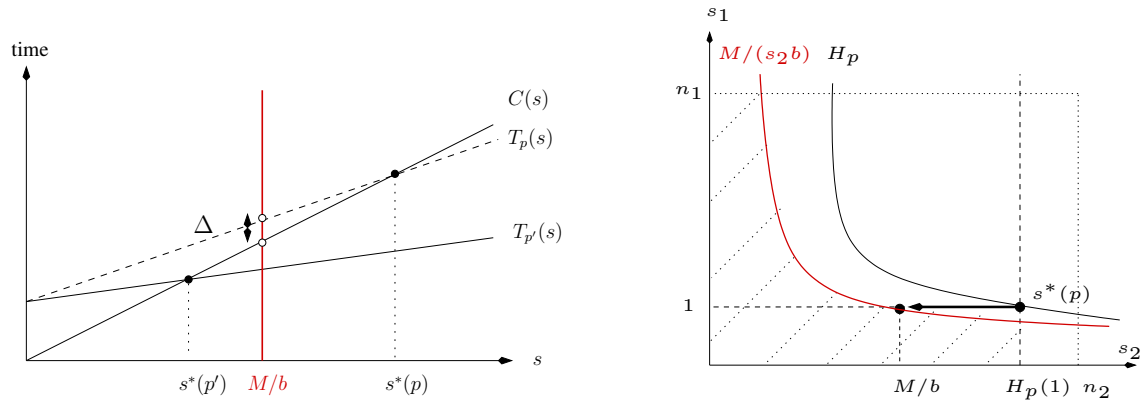
So far, we reasoned about optimal granularity only based on the ratio between computation time and transfer time assuming no local memory constraint. In the following, we discuss this issue.

One-dimensional data:

The local memory size constraint, which imposes an upper bound M on the size of a buffer, is simply represented by the granularity $s = M/b$.

Given p processors, if optimal granularity does not fit in local memory, that is $s^*(p) > M/b$ as illustrated in Figure 3.8-(a), then near optimal solutions have to be considered. We can think of two possible candidates presented as a combination of number processors and granularity,

1. $(p, M/p)$: keeping the same number of processors and decreasing the granularity to the maximum possible value.



(a) One-dimensional data

(b) Two-dimensional data

Figure 3.8: Local memory constraint.

2. $(p', s^*(p'))$: reducing the number of processors $p' < p$ involved in the computations to reduce the corresponding optimal granularity $s^*(p')$. For this, we need to identify the maximum value of p' such that $s^*(p') \leq M/b$, a fact characterized by $\alpha_p \leq (\omega/b) - (I/M)$ (or $p \leq (\omega/\alpha b) - (I/\alpha M)$ if $\alpha_p = \alpha \cdot p$).

Which solution gives better performance? well, using more processors has the advantage of reducing the quantity of work per processor and the number of iterations m, however since M/b is in the transfer domain, at each iteration processors are idle $\Delta = T_p(M/b) - C(M/b)$ time units waiting for data. The answer to the question then depends on the difference Δ, $p - p'$ and $M/b - s^*(p')$.

Two-dimensional data:

The local memory size constraint $s_2 s_2 b \leq M$ is proportional to the area of a block $s_1 \times s_2$. More generally, for any point $s = (s_1, s_2)$, the area of the rectangle defined by its coordinates represents the memory capacity required for this granularity. Local memory constraint is then represented by the hyperbola shown in Fig. 3.8(b), excluding solutions (shapes) above the hyperbola that do not fit in local memory.

For a given value of p, if the optimal solution $(1, H_p(1))$ does not fit in local memory, one can think of other points of the hyperbola H_p which are, as explained previously, candidates for optimality since they satisfy the equality $T_p = C$. However, we know that as we move upwards in the hyperbola H_p, the transfer time increases and consequently computation time which is proportional to the area. Therefore the point $(1, H_p(1))$ is also the point with minimal transfer time as well as computation time and area (means that it is the closest point of the hyperbola H_p to the origin). This proves that if this point does not fit in local memory then no other point (shape) of the hyperbola fits in local memory, which is viewed as the hyperbolas $H_p(s_1)$ and $M/(s_1 b)$ do not intersect.

Like the one-dimensional case, the two candidates for near optimal solutions are the pairs, $(p, (1, M/b))$ and $(p', (1, H_{p'}(1))$ where $p' < p$.

3.3.4 Conclusion

In this chapter we focused on deriving optimal granularity for independent data computations where the main idea is that the ratio between computation time and transfer time

at each iteration of the pipeline varies with the block size and shape, we can therefore optimize performance by finding the right balance between computations and data transfers to hide completely the memory latency while minimizing the prologue and epilogue of the pipeline. This issue is closely related to the number of processors involved in the computations which changes this ratio by increasing the transfer time, due to contentions, thus requiring higher granularities to optimize performance.

In terms of performance, the main significant switch is between a transfer regime and a computation regime, which is clear in the one-dimensional case as it occurs beyond some value s^*, but is much less clear in the two-dimensional case. Note that there is a limit of how much performance can be improved by varying the granularity before the execution becomes computation dominant and where there is no point in increasing the granularity.

The local memory limitation is a strong hardware constraint in MPSoCs, if optimal granularity does not fit in the memory limitations then finding the right combination of number of processors and data granularity is required.

Chapter 4

Shared Data Transfers

4.1 Introduction

The second part of our contribution is dedicated to *overlapped* data computations as defined in Section 2.3.2. Like independent data computations, the input array is partitioned into blocks of data and a double buffering algorithm is implemented in order to overlap computations and data transfers, however as illustrated in Figure 4.1 these blocks share data. We focus first on one-dimensional data where the size of shared data is fixed, so besides deriving optimal granularity, we compare several strategies for transferring shared data. We then focus on deriving optimal granularity for two-dimensional data where the size of shared data varies according to the geometry of partitioning thus influencing the choice of block shapes.

4.2 Transferring Shared Data in one-dimensional data

We assume that neighboring data required for the computation of a basic block constitute a window of k basic blocks situated to the left [1], where $k << n$ to keep some locality in the computations. Such patterns are common, for example, in signal processing application with overlapping time windows. Note that regardless of granularity for super blocks, the size of shared data is always fixed due to the one-dimensional geometry of the array.

In the following, we consider three strategies/mechanisms for transferring shared data that mainly differ in the component of the architecture which carries the burden of the

1. The same reasoning can be applied to the right direction or both left and right directions.

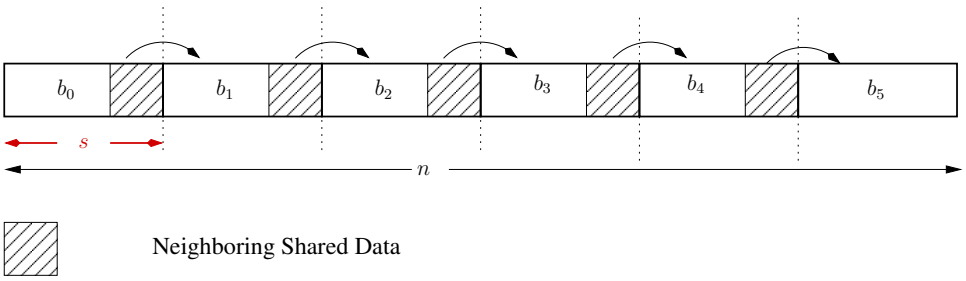


Figure 4.1: Shared data between neighboring blocks in a one-dimensional array.

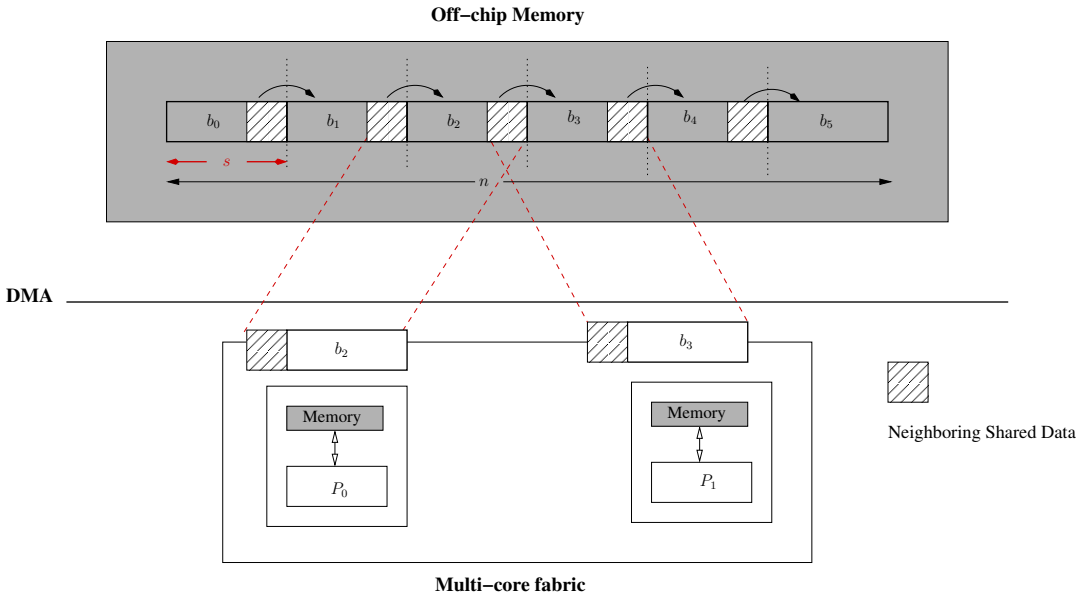


Figure 4.2: Replication of shared data.

transfer,

1. *Replication*: transfer via the interconnect between local and off-chip memory;
2. *Inter-processor communication*: transfer via the network-on-chip between the cores;
3. *Local buffering*: transfer by the core processors themselves.

For each strategy, we construct an analytical performance model used to derive optimal granularity and then compare their performance. These models are useful to give some guidelines to the programmer/compiler for choosing a data sharing strategy along with the granularity of data transfers. In the following, we detail and discuss these strategies.

4.2.1 Replication

Given a granularity s, the input array X is divided into super blocks of s basic blocks each, allocated to processors in a periodic fashion for computations. For each transfer of a super block from off-chip memory to a local memory, additional neighboring data is also transferred. Neighboring blocks are mapped to neighboring processors and the *same* data is *replicated* in each of the processor's local memory, as illustrated in Figure 4.2.

The main advantage of this strategy is that computations among processors become completely independent and no synchronization between them is required. The analysis is similar to the previous chapter where optimal granularity is about finding the right balance between the computation and the transfer times to hide completely the memory latency. Thus considering at each iteration the overhead of transferring k additional basic blocks, optimal granularity is reached for the granularity s^* so that,

$$T(s^* + k,\ p) = C(s^*)$$

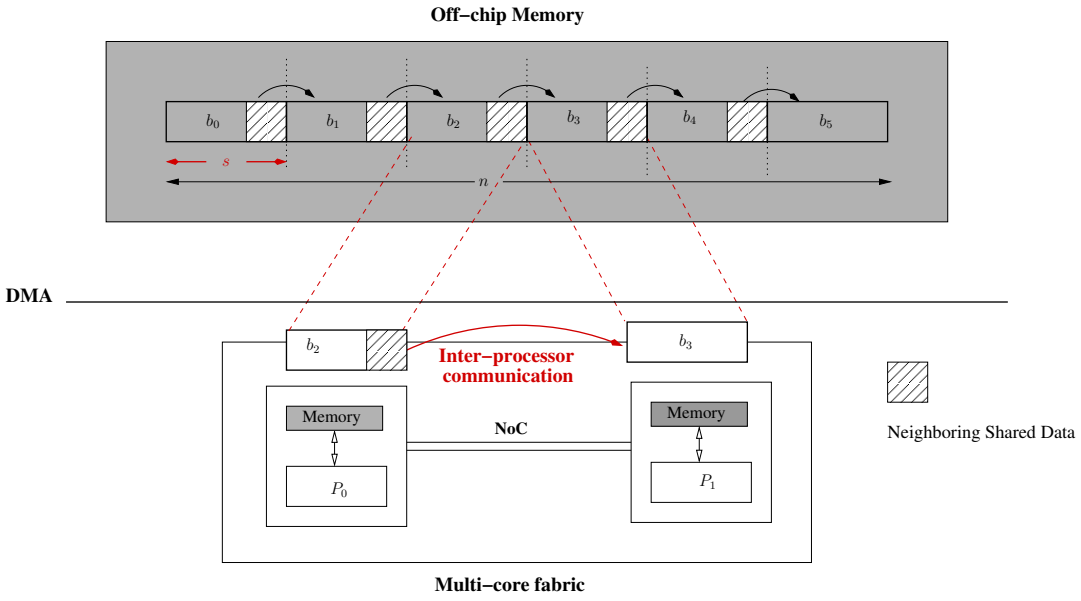


Figure 4.3: Communication of shared data between neighboring processors.

and the overall execution time is,

$$\tau(s,p) = \begin{cases} (n/sp+1)\,T(s+k,\ p) & \text{for } s < s^* \\ 2\cdot T(s+k,\ p) + (n/p)\omega & \text{for } s > s^* \end{cases}$$

4.2.2 Inter-Processor Communication

The main idea in this strategy is to transfer data blocks from off-chip memory to processors local memory without replication and then the processors can exchange required shared data via the high speed local interconnect as illustrated in Figure 4.3. For this, we need to allocate data blocks to processors in a *periodic* rather than contiguous fashion so that neighboring blocks are to be processed by neighboring processors.

The behavior of the pipelined execution is illustrated in Figure 4.4. As for independent data using multiple processors, transfers from external memory are performed concurrently to overlap the initialization step of DMA commands. After reception of a super block each processor p_j sends the overlapping data to its right neighbor p_{j+1}. Processors stall at that time waiting for the end of the communication. Prefetching next blocks can then be issued to be done concurrently with computations. This execution assumes the ideal case where the communication geometry matches the geometry of the local interconnect thus allowing processors to perform in *parallel* a point to point inter-processor communication. In this case, data allocation constitutes a ring-like topology which matches the hardware topology of the local interconnect in our experiments.

To reason about optimal granularity, we characterize the inter-processor communication cost for transferring shared data, a block of size $b \cdot k$ bytes, from one processor's local memory to another. This cost, that we denote by R_k, obviously depends on the transfer mechanism used: load/store instructions via a shared memory location, explicit DMA commands, etc. We assume that inter-processor communication is also performed using DMA. In terms of performance, they have the same initialization cost I, and a transfer cost per byte $\beta \ll \alpha$ because the network-on-chip connecting the cores as well as their

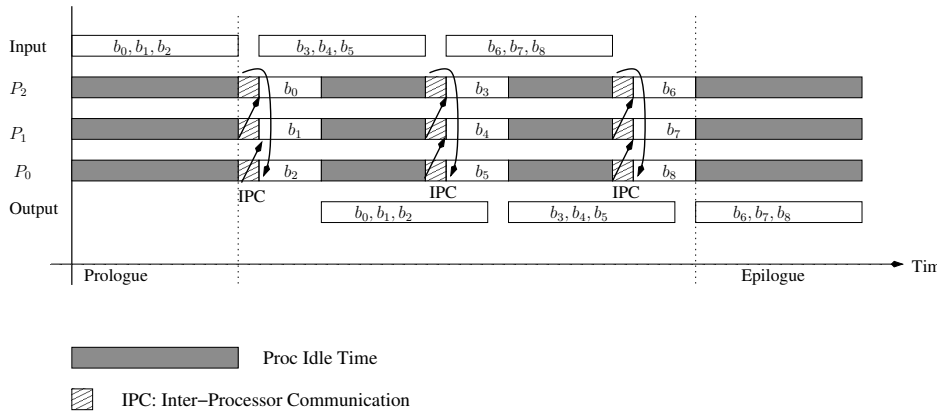


Figure 4.4: Pipelined execution using multiple processors with inter-processor communication.

memories (usually SRAMs) is much faster. Thus R_k is approximately by,

$$R_k = I + \beta \cdot b \cdot k$$

In practice, this cost also includes the synchronization overhead between the cores which can be included in the value of β or expressed with additional terms that depend on p.

In our case, we consider the use of *blocking* DMA calls for inter-processor communication which means that processors are idle for R_k time waiting for the transfer completion. This overhead is added to the computation time, and since our main focus is to hide the off-chip memory latency, optimal granularity is reached for s^* so that,

$$T(s^*, p) = C(s^*) + R_k$$

The overall execution time is,

$$\tau(s,p) = \begin{cases} (n/sp + 1)\, T(s,p) & \text{for } s < s^* \\ 2 \cdot T(s,p) + (n/p)\omega + (n/sp)R_k & \text{for } s > s^* \end{cases}$$

Note that for the sake of simplicity, we assume that $k < s$ to restrict inter-processor communication to the immediate neighbors.

Also, note that one can also consider double buffering to hide inter-processor DMA communication, we assume that off-chip memory latency is the main bottleneck for performance since both the local interconnect latency and the size of shared data are much smaller.

4.2.3 Local Buffering

When input array X is partitioned into p contiguous chunks allocated to processors where each chunk is also partitioned into super blocks of s basic blocks each, then neighboring blocks are allocated to the same processor and computed at successive iterations. This strategy exploits this fact so that shared data is stored in the processor's local memory since it will be required for computations at the next iteration.

In terms of performance, keeping shared data in the local memory is not for free, since at each iteration, this data has to be copied from one local buffer to the other using

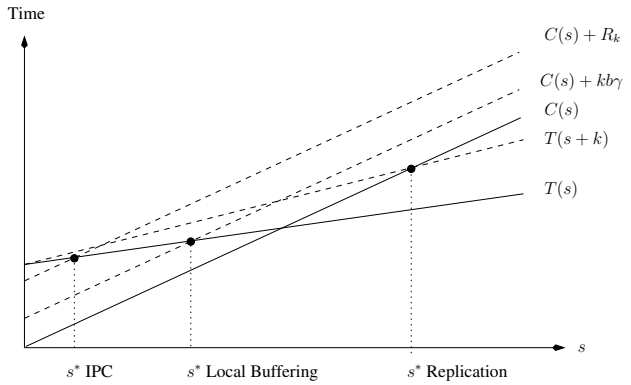


Figure 4.5: Comparing optimal granularity between strategies.

load/store instructions [2]. We characterize the local copying overhead with $k \cdot b \cdot \gamma$ where γ is the cost per byte of a load/store instruction.

Since k is fixed, this overhead is also fixed and it is added at each iteration to the computation time. Optimal granularity is obtained at s^* so that

$$T(s^*, p) = C(s^*) + k \cdot \gamma$$

and the overall execution time $\tau(s, p)$ is,

$$\tau(s,p) = \begin{cases} (n/sp + 1)\, T(s,p) & \text{for } s < s^* \\ 2 \cdot T(s,p) + (n/p)\omega + (n/sp)(k \cdot \gamma) & \text{for } s > s^* \end{cases}$$

4.2.4 Comparing Strategies

The models defined above for each strategy involve, on one hand, parameters of the application: b, k and ω (assuming processor speed is fixed) and parameters of the architecture: α, β, γ. According to these models one can optimize the decision variables which are p, s and the sharing strategy.

Local buffering and inter-processor communication schemes have a clear computation overhead due to the time the processor spends at each iteration copying data from one local buffer to another, or idle waiting for the completion of the inter-processor *synchronous* DMA communication. On the other hand, replication contributes to the increase in the off-chip memory latency because more data needs to be transferred. As discussed previously, this overhead becomes more significant when the number of processors increases.

Therefore, for the same instance of the problem, replication switches to the computation regime at a higher granularity than local buffering and inter-processor communication, see Figure 4.5, since off-chip memory latency is higher. However, when all three strategies reach their computation regime, replication always performs better because of the processing overhead; $(n/sp)R_k$ or $(n/sp)(k \cdot \gamma)$, which corresponds to the time the processor spends copying shared data locally, or communicating data to other processors. Note that this overhead is proportional to the number of iterations in the pipeline (n/sp) and is therefore reduced as s or p increase, leading to nearly similar performance results among strategies.

2. More efficient pointer manipulation is hard to implement because the same buffer is used to store $x[i-1]$ and $x[i+1]$ which is filled *in parallel* with the computation of $y[i]$.

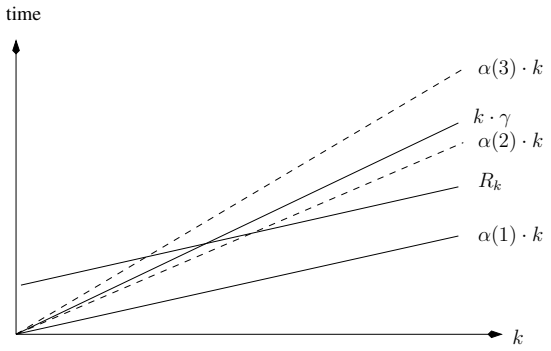


Figure 4.6: Comparing parameters for strategies.

When transfer time is dominant, comparing the strategies boils down to comparing the cost of transferring the *additional* shared data using different hardware mechanisms: $k \cdot \alpha(p)$, $I + k \cdot \beta$ and $k \cdot \gamma$. Figure 4.6 illustrates the sensitivity of these quantities to k and p. In this example, for one processor, replication cost is lower than local copying and inter-processor communication, but as the number of processors increases, the overhead of contentions while accessing off-chip memory ruins the performance compared to the other strategies, where the transfer of shared data is delegated to the high speed network-on-chip or to the processors and is totally or partly done in parallel.

Memory limitation can play a role in the choice of optimal strategy when optimal granularity for the replication strategy does not fit in local memory, the parameters comparison can then give a hint about the strategy that gives best performance given the available memory space budget.

4.3 Optimal Granularity for Two-Dimensional Data

For two-dimensional data arrays we consider the neighborhood pattern defined in section 2.3.2 where neighboring data required for computations constitute a symmetric window of size k. Given this pattern, we want to derive optimal granularity considering shared data, first for a single processor and then multiple processors.

We assume a *replication strategy* for transferring shared data where additional data is transferred at each iteration, along with the block to compute, from off-chip memory to local memory. Under this assumption, the constrained optimization problem to derive optimal granularity, defined in (3.4), becomes

$$\begin{aligned} &\min\ T(s_1 + k, s_2 + k)\ \text{ s.t.} \\ &\\ &T(s_1 + k, s_2 + k) \leq C(s_1, s_2) \\ &(s_1, s_2) \in [1, n_1] \times [1, n_2] \\ &b \cdot (s_1 + k) \cdot (s_2 + k) \leq M \end{aligned} \tag{4.1}$$

Recall that without data sharing, optimal granularity is attained by the flat block which reaches the computation regime and optimizes the DMA latency overhead per line and thus the prologue and epilogue. However, as discussed in section 2.3.2, the quantity of replicated data: $k(s_1 + s_2) + k^2$ which varies according to the block shape is minimal for *square* blocks, among all the super blocks of the same area. Therefore, these two facts are

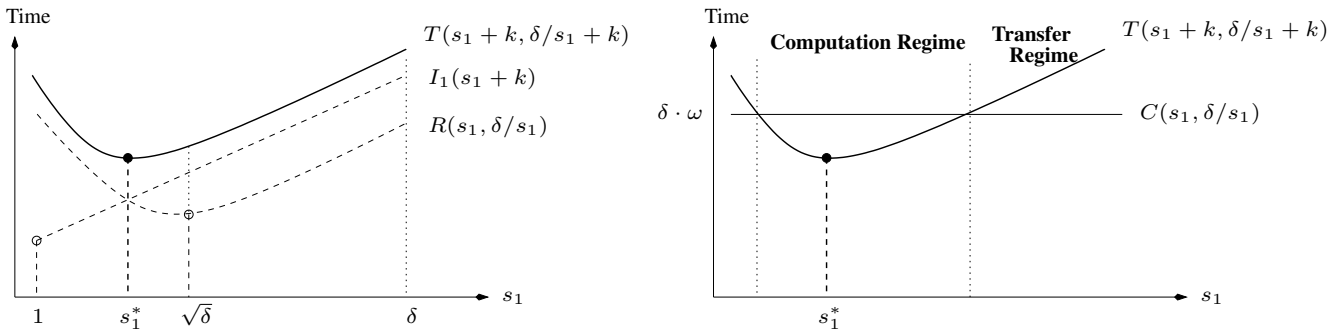


Figure 4.7: DMA transfer cost with replicated area as we increase the number of lines in a block, for a given $\delta = s_1 \times s_2$.

in conflict and when they are combined there is a balance to be found between the two. This excludes for optimality the two extreme solutions and justifies the analysis to find the optimal granularity which is somewhere between flat and square shapes.

In the sequel we first explain how the shape of the block and its replicated area influence the transfer cost and then derive optimal shape.

Replicated Area and Transfer Cost

To process a super block of shape $s_1 \times s_2$, one needs to load

$$R(s_1, s_2) = (s_1 + k) \times (s_2 + k)$$

basic blocks. The DMA transfer cost of a rectangular super block considering replication of shared data is therefore,

$$T(s_1 + k, s_2 + k) = I_0 + I_1(s_1 + k) + \alpha b \cdot R(s_1, s_2) \tag{4.2}$$

Figure 4.7-(a) illustrates this function for a *fixed* value of $\delta = s_1 \times s_2$ along with the DMA issue time overhead optimized for flat block transfer ($s_1 = 1$) and the replicated data transfer overhead optimized for square shapes $(\sqrt{\delta}, \sqrt{\delta})$. Among all combinations (s_1, s_2) satisfying $s_1 \times s_2 = \delta$, the transfer cost is minimal for the point $(s_1^*, \delta/s_1^*)$ where $s_1^* = \sqrt{\alpha bk\delta/(I_1 + \alpha bk)}$. This point represents *the* balance between initialization phase overhead (number of lines) and transfer phase cost (amount of replicated data).

Note that if we look at the computation time of these blocks, then all shapes satisfying $s_1 \times s_2 = \delta$ have approximately the same computation time: $\delta \cdot \omega$, proportional to the area. According to the balance between a block computation time and its transfer time, some shapes will lead to a computation regime and others to a transfer regime as depicted in Figure 4.7-(b). The point $(s_1^*, \delta/s_1^*)$ is then the optimal granularity for each value of δ (assuming $\delta \cdot \omega \geq T(s_1^* + k, \delta/s_1^* + k)$) since it minimizes the transfer time.

In the following we derive optimal granularity for all shapes yielding a computation regime.

Optimal Granularity

Optimal granularity is the point $s^* = (s_1^*, s_2^*)$ in the computation domain which optimizes DMA transfer time, in order to minimize both the prologue and the epilogue of the pipeline.

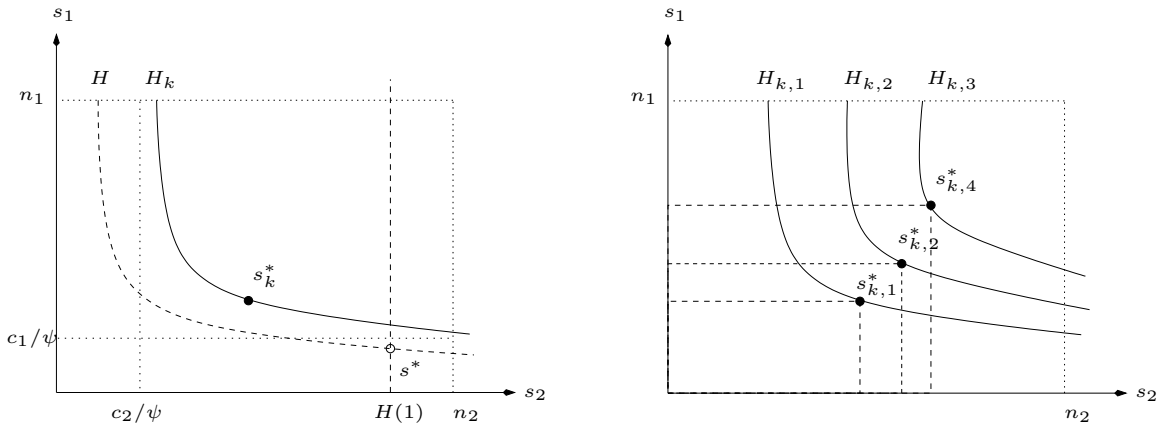


Figure 4.8: Computation domain and optimal granularity considering replication of shared data: (a) Single processor, (b) As we increase the number of processors.

As for independent computations, the computation domain is defined by the inequality $T(s_1 + k, s_2 + k) \leq C(s_1, s_2)$ and because this domain is convex, then candidates for optimal granularity are restricted to the points $(s_1, H_k(s_1))$ satisfying the equality $T = C$. The problem is reduced to minimizing $T(s_1, H_k(s_1))$ where,

$$H_k(s_1) = (c_2 s_1 - c_3)/(\psi s_1 - c_1)$$

c_1, c_2 and c_3 are positive integer constants that depend on I_0, I_1, α, b and k such that,

$$\begin{cases} c_1 = \alpha b k \\ c_2 = c_1 + I_1 \\ c_3 = I_0 + I_1 k + \alpha b k^2 \end{cases}$$

$T(s_1, H_k(s_1))$ is a second order function with one variable. By computing the derivative, we get one negative point that is not interesting for us and another positive point that is the optimal.

To simplify the reading of the formulas, let $\Delta = (c_1/\psi)[1+D]$ where $D = \sqrt{c_3\alpha/c_1 c_2}$, then optimal granularity is the point $s^* = (s_1^*, s_2^*)$ so that,

$$\begin{cases} s_1^* = \Delta + (c_1/\psi)(1/D) \\ s_2^* = \Delta + (I_1/\psi)(1 + D) \end{cases}$$

Fig. 4.8-(a) illustrates the evolution of the computation domain and the optimal granularity while considering shared data. As discussed in the previous section, we can clearly see that optimal granularity is somewhere between a flat and a square block as s_1^* and s_2^* are both equal to Δ plus a different offset each.

As we consider multiple processors, recall that we parametrize the transfer cost per byte α with the number of active processors such that α_p increases monotonically with p, in order to model contentions due to concurrent processors transfer requests. We use T_p to denote DMA transfer time with α_p replacing α.

The reasoning is similar to previously where function H becomes H_p yielding an optimal shape for each p. Figure 4.8-(b) illustrates the evolution of the computation domain and optimal granularity as we increase the number of processors. As for independent data computations, the computation domain is reduced as we increase the number of processors since the ratio between computation time and transfer time is also reduced. Also,

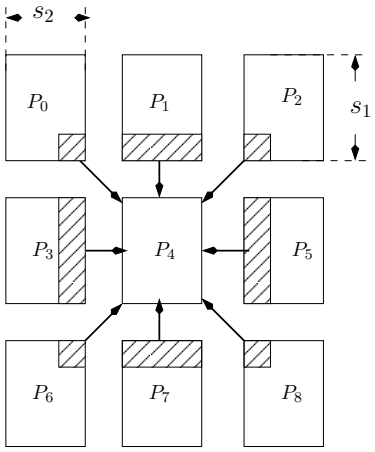


Figure 4.9: Neighboring data communication for two-dimensional data.

optimal granularity increases with the number of processors in order to keep the processors busy enough, the time to fetch data required by the processors for next iteration.

Other Strategies for Transferring Shared Data

As for one-dimensional data, one can think of other strategies than replication for transferring shared data such as inter-processor communication, where at each iteration each processor fetches a super block of (s_1, s_2) basic blocks and then neighboring processors exchange shared data, as illustrated in Figure 4.9. In practice, this is hard to program and in terms of performance it induces a large synchronization overhead since 8 processors are required to implement the geometry of the defined neighborhood pattern. Furthermore, it also induces a large communication traffic and overhead especially if the program topology does not match the underlying hardware topology.

One can also think about combining, in the same implementation, several strategies (inter-processor communication or local buffering with replication) where a different technique is used to transfer neighboring data in each direction. Figure 4.10 shows how inter-processor communication can be combined with replication. In (a) [3], neighboring horizontal data is exchanged between processors as at each iteration a super block of $(s_1 + k, s_2)$ basic blocks is fetched from main memory. k shared basic blocks are in the sequel exchanged with right and left neighboring processors, thereby requiring a minimum number of 3 concurrent processors. Note that in this case, data partitioning also combines contiguous and periodic processors data allocation.

Horizontal inter-processor communication, as it is illustrated in Figure 4.10-(a), requires copies of *non contiguous data* (additional data per line) from one processor's local memory to another. This induces a significant overhead as it requires a separate DMA call for each contiguous transfer (strided DMA transfers are not possible between two processors local memory) or an extra processing overhead for reorganizing strided data in local memory in a contiguous fashion.

Vertical inter-processor communication seems an interesting option to investigate since DMA is more efficient for transferring contiguous data blocks, which makes the cost of

3. We explain for (a) and it is the same for (b).

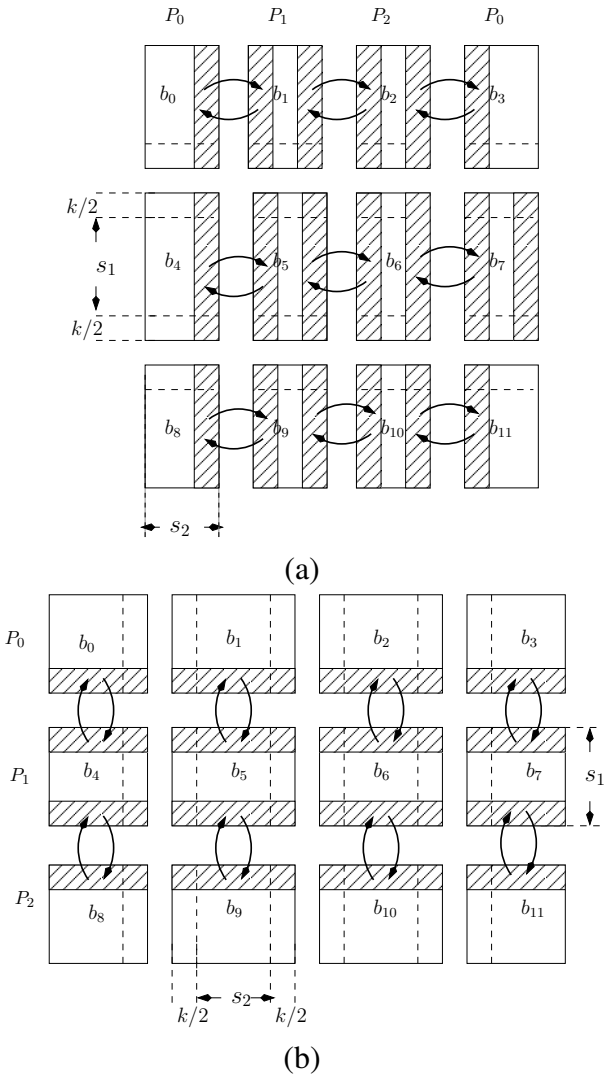


Figure 4.10: Combining replication and inter-processor communication in different directions, (a) horizontal data exhange, (b) vertical data exchange. Shaded areas correspond to shared data.

replicating additional data per line negligible. Also, it avoids the cost of transferring additional data lines from main memory, these contiguous lines are efficiently transferred using DMA between two processors local memory.

In this thesis, we focused only on replication as other strategies are in some cases hard to implement and induce a significant overhead thus making them less interesting.

4.4 Conclusion

In this chapter, we focused on deriving optimal granularity for applications that share data where the main focus remains to find the right balance between computations and data transfers, based on hardware and software parameters.

For one-dimensional data, we compared several mechanisms for transferring shared data. For each mechanism, we characterize the cost of transferring shared data, we derive

optimal granularity and approximate the overall performance in the computation and the transfer regimes. Comparing the strategies obviously depends on the hardware parameters that characterize the platform but more importantly on the size of shared data and the number of processors involved.

Two-dimensional data structures with data sharing is the most interesting case which justifies the most the analysis, as the optimal shape is hard for programmers to intuitively guess. They usually pick trivial choices: contiguous or square shapes, even though they are aware of the influence of the shape of the block on the size of shared data and the DMA preference for contiguous blocks. These aspects becomes hard to combine especially as further the balance between computation time, which also depends on the shape, has to be considered.

Chapter 5

Experiments

5.1 Introduction

In this chapter, we validate the simplified models presented earlier on a real architecture, the Cell Broadband Engine Architecture, represented by a cycle-accurate simulator, and check whether their predictive power is sufficiently good to serve as a basis for optimization decisions.

In the sequel, we give an overview of the Cell B.E. architecture and derive, based on profiling of the architecture, the values of the hardware parameters required for the analysis. Based on these values we then derive optimal granularity for double buffering algorithms first for independent data computations and then for applications that share data.

5.2 Cell BE

5.2.1 Overview

The Cell Broadband Engine Architecture is a 9-core heterogeneous multi-core architecture, consisting of a Power Processor Element (PPE) linked to 8 Synergistic Processing Elements (SPE) acting as co-processors, through internal high speed Element Interconnect Bus (EIB) as shown in Figure 5.1. The PPE and SPE processors run at 3.2GHz clock frequency and the interconnect is clocked with half the frequency of the processors.

The PPE is a 64-bit PowerPC Processor Unit (PPU) and each SPE is composed of a Synergistic Processing Unit (SPU) which is a vector processing unit, an SRAM local store (LS) of size 256 kbytes shared between instructions and data, and a Memory Flow Controller (MFC) to manage DMA data transfers. The PPU provides a single shared address space across SPEs and the MFC's memory translation unit handles the required address translation. An SPE can access the external DRAM and the local store of other SPEs only by issuing DMA commands. The PPU can also issue DMA commands supported by the MFC. The MFC supports aligned DMA transfers of 1, 2, 4, 8, 16 or a multiple of 16 bytes, the maximum size of one DMA transfer request being 16K. To transfer more than 16K, DMA lists are used. Further details of the architecture can be obtained at [IBM08].

As mentioned in Section 2.2, the MFC of each SPE is composed of a Direct Memory Access Controller (DMAC) to process DMA commands queued in the MFC and of a

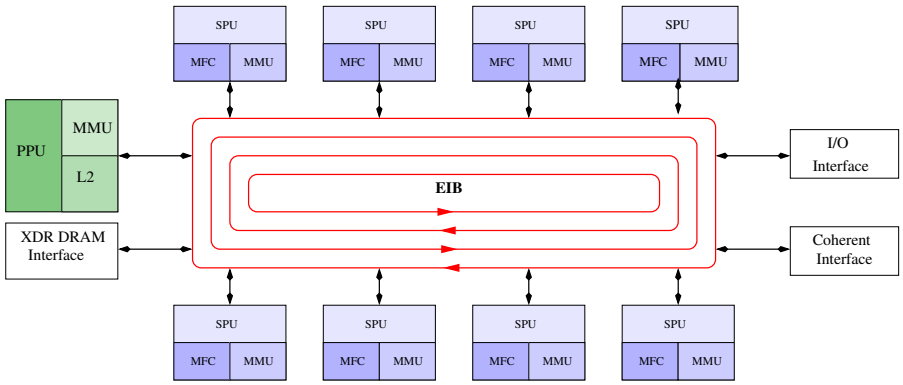


Figure 5.1: Cell B.E. Architecture

Memory Management Unit (MMU) to handle the address translation required by the selected DMA command. The MMU has a translation look-aside buffer (TLB) for caching recently translated addresses. TLB misses can affect significantly the performance, therefore we neglect this effect by doing a warm-up run which will load the TLB entries before profiling of the program. Also, we allocate large page tables to have a smaller number of TLB entries for data array. As explained previously, after the address translation, the DMAC splits the DMA command into smaller bus transfers. Peak performance is achievable when both the source and destination address are 128-byte aligned and the block size is multiple of 128 bytes [KPP06]. We can observe reduction in performance when this is not the case.

For our experiments, we use a Cell B.E. simulator [IBM09] whose performance are very close to the actual processor. However, it does not model precisely all the main memory details. In the Cell B.E. references, it is mentioned that it models a DDR2 memory instead of RAMBUS XDR, used in the real processor, and it does not have Replacement Management table for cache. This does not affect much the validity of our results since we do not consider either a detail description of the main memory and performance is measured directly on the SPUs.

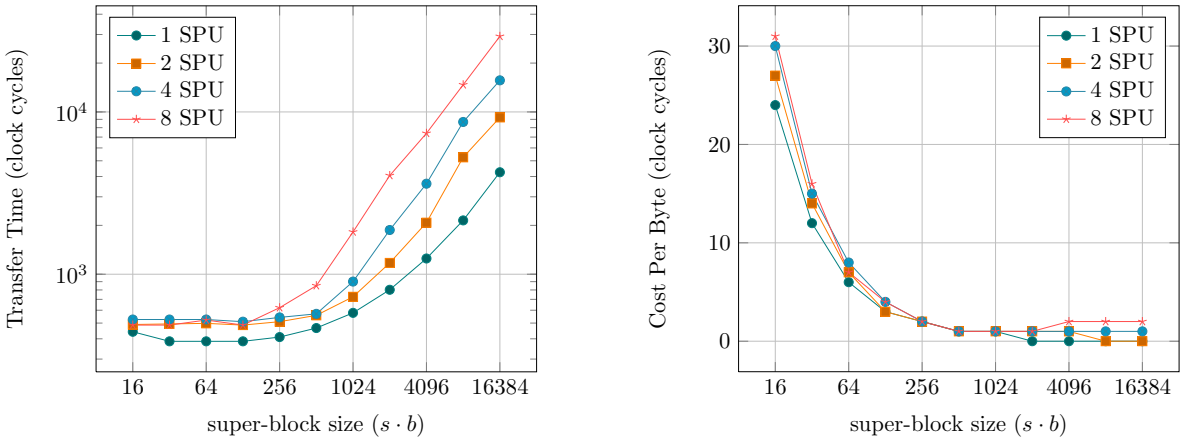
For programming the Cell B.E. platform, we use a low level library provided by the Cell B.E. SDK [IBM08]. This library gives us direct control over the platform and excludes any overhead due to the implementation of high level constructs used by a high level programming model. However, the low level library exposes to the programmer all the low level hardware details which requires the programmer to have a good understanding of the architecture and its hardware constraints, to provide an efficient implementation. One example of such hardware constraints is data alignment issues.

In the following, using profiling we retrieve the values of the hardware parameters of the Cell B.E. platform required for the analysis.

5.2.2 Hardware Parameters Measurement

To measure the DMA latency, we implement small benchmarks issuing blocking DMA requests and we measure the DMA transfer time as we vary the block size and shape and increase the number of processors issuing concurrent transfer requests.

The DMA transfer time for one-dimensional blocks and accordingly the cost per byte are plotted in Figure 5.2. As mentioned previously, the DMA has a large initialization



(a) DMA transfer time

(b) DMA Read Cost per Byte

Figure 5.2: DMA performance for contiguous blocks

	α_p		
p	min	max	avg
1	1.13	14.00	2.57
2	1.78	29.98	4.13
4	3.97	47.23	11.07
8	5.43	87.86	18.82

Table 5.1: The transfer time per byte for two-dimensional data transfers, as a function of the number of processes.

overhead, the measured initialization phase time I is about 400 cycles, which is amortized for block size larger than 128 bytes since the DMA cost per byte is reduced significantly for this value. As we increase the number of processors and synchronize concurrent DMA transfers, we can observe that the transfer time is not highly affected for a small granularity because the initial phase of the transfer is done in parallel in each processor's MFC, whereas for a large granularity the transfer time increases proportionately to the number of processors due to the contentions of concurrent requests on the bus and bottleneck at the Memory Interface Controller (MIC) as explained in [KPP06]. The measured DMA transfer cost per byte to read from main memory $\alpha(1)$ is about 0.22 cycles per byte, it increases proportionately to the number of processors to reach $p \cdot \alpha(1)$ (for large granularity transfers).

For two-dimensional data blocks DMA transfers, implemented using DMA lists, we also derive the DMA parameter values based on profiling information. As modeled, these values consist of a fixed initialization cost $I_0 = 108$ and an initialization cost per line $I_1 = 50$ cycles which corresponds to the cost of the creation of each list element.

The transfer cost per byte α is subject to variations that are more visible and amplified for rectangular data block transfers. These variations are due to several factors such as concurrent reading and writing requests of the same processor, packet-level arbitration between requests of different processors as well as the effect of strided accesses in main memory. The minimal, maximal and average values of α_p measured for two-dimensional data are shown in Tab. 5.1 and we use the average value in our model.

Note that due to the characteristics of the Cell B.E. not all block size and shape com-

binations are possible. Indeed a DMA list can hold up to 2K transfer elements. Each element is a contiguous block transfer with maximum size 16KBytes. Furthermore, the Cell B.E. has a strict alignment requirements on 16-byte boundary for both DMA transfers and SPU vector instructions for which the processor is optimized. If this is not taken care of, the DMA engine aligns the data by itself causing erroneous results.

5.3 Experimental Results

For our experiments, we implement a double buffering algorithm for different benchmarks, some of them are applications where computations are completely independent and others share data. For each benchmark, we run the experiments as we vary the block size and shape and the number of processors and check whether the analytical optimum is close to the measured one and the measured performance close to the predicted one.

Our benchmarks consist of, first synthetic algorithms of (independent/shared) computations where f is an abstract function for which we vary the computation workload per byte ω and the size of shared data.

We then implement a convolution algorithm that computes an output signal based on an input signal and an impulse response signal. These signals are encoded as one-dimensional data arrays and the size of the impulse signal determines the size of the data window required to compute one output item. We vary the size of the impulse signal to vary the size of the neighboring shared data.

Our last benchmark is a mean filtering algorithm working on a bitmap image, encoded as a two-dimensional data array. This algorithm computes the output for each pixel as the average of the value of its neighborhood.

5.3.1 Independent Computations

To validate our analytical results for independent computations, we use synthetic algorithms and focus only on one-dimensional data since, as explained previously, optimizing data granularity for independent two-dimensional data is similar to the one-dimensional case where it is about optimizing the size (and not shape) of data.

For synthetic algorithms, we fix the size b of a basic block to 16 bytes and the size n of input data array to 64K, the total size of the array being 1Mbytes. Also we keep the maximal number of threads that are spawn on SPUs equal to the number of SPU's. This will avoid context switching and gives more predictable results. Also there will be no question of scheduling which is another part of the problem and not in the scope of this thesis. The memory capacity limits then the possible number of blocks clustered in one transfer to $\bar{s} < 4K$, excluding memory space allocated to code size. We vary the number of blocks s and the number of processors. We compare both predicted and measured optimal granularity, and the total execution time for both transfer and computation regimes. Figure 5.3 shows the predicted and measured values for 2, 4 and 8 processors. We can observe that the values are very close to each other. The predicted optimal points are not exactly the measured ones but they give very good performance. Performance prediction in the computation regime is better than in the transfer regime, because the computations which constitute the dominant part of the total execution time is performed on processors. Besides, as mentioned in [SBKD06] we have sufficient time to hide delays caused due to the network latency and bandwidth.

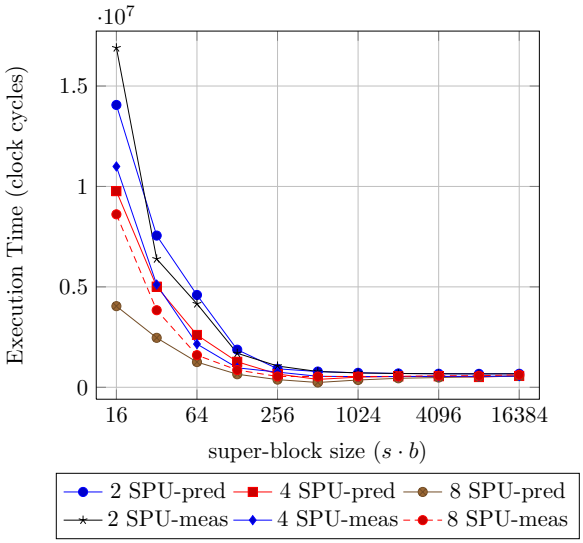


Figure 5.3: Independent data computations

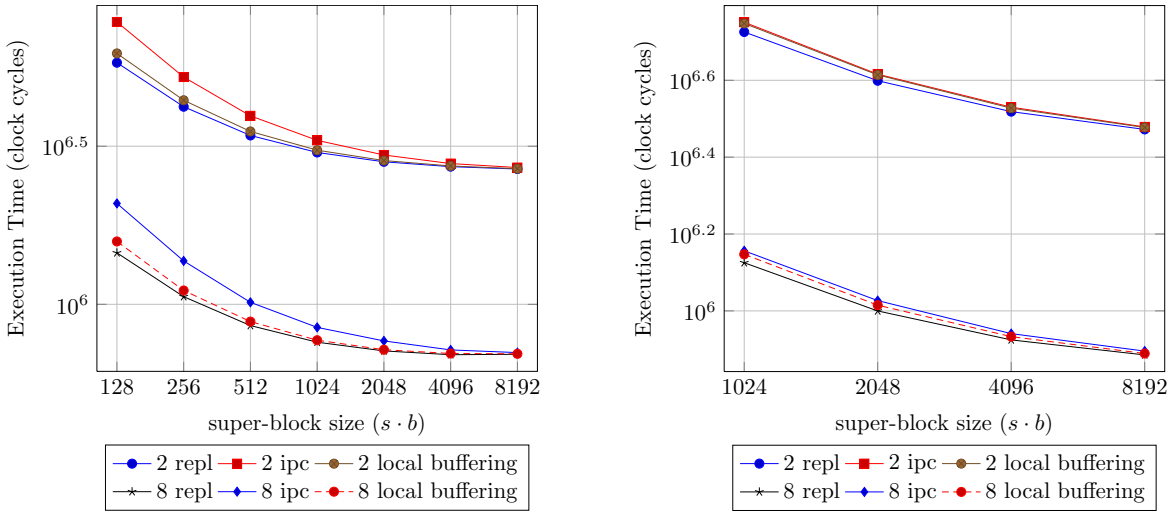
5.3.2 Shared Computations

5.3.2.1 Synthetic Benchmarks

We implement the different strategies for transferring shared data explained in Section 4.2. We run experiments with different values of k, s and p, for both computation and transfer regimes by varying the computation workload ω. We present the results for a small and a large value of k, 128bytes and 1K respectively, and for 2 and 8 processors.

In Inter-Processor Communication (IPC) strategy, we make sure that neighboring processors exchanging data are physically mapped close to each other. Specifying affinity during thread creation in linux allows the logical threads to be mapped physically next to each other. This gives advantage of having higher bandwidth as mentioned in [SNBS09]. The global data partitioning specified in section 4.2.2 has a ring geometry of communication similar to the EIB topology, so that each processor can send data to its right neighbor. The processors must synchronize with each other to send shared data after the DMA request for fetching a super-block from main memory has completed. We experiment with two variants of IPC synchronization: a point to point signaling mechanism to check the availability of shared data and acknowledge the neighbor, and a global barrier based on signals mentioned in [BZZL08] to synchronize all processors. Because of the high synchronization overhead compared to point to point signaling, we do not present here the results obtained with this variant. After processors synchronization, the transfer of shared data is done by issuing a DMA command.

As discussed in section 4.2.4, in the computation regime replication performs always better than local buffering and IPC as shown in Figures 5.4a and 5.4b. Besides, we can see that IPC performs worse than local buffering because the cost per byte γ via load/store operations which is around 2 cycles per byte, is much lower than the cost R_k to perform IPC synchronization and DMA calls. R_k involves a DMA inter-processor cost per byte β and a synchronization cost. In practice, it is very difficult to estimate R_k precisely, mainly because of the difficulty in predicting the exact arrival time in the presence of continuous read/write data-transfers. In our experiments, the initial time of an inter-processor DMA command was around 200 cycles, the transfer cost per byte β around 0.13 cycles and the



(a) shared data size 128 bytes

(b) shared data size 1024 bytes

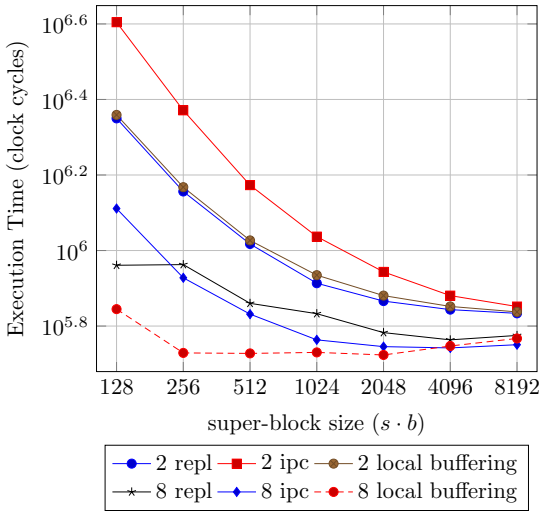
Figure 5.4: Overlapped data in computation regime

synchronization cost between 200 and 500 cycles as we vary the number of processors. The synchronization cost using barrier is much larger, between 800 and 2400 for each iteration.

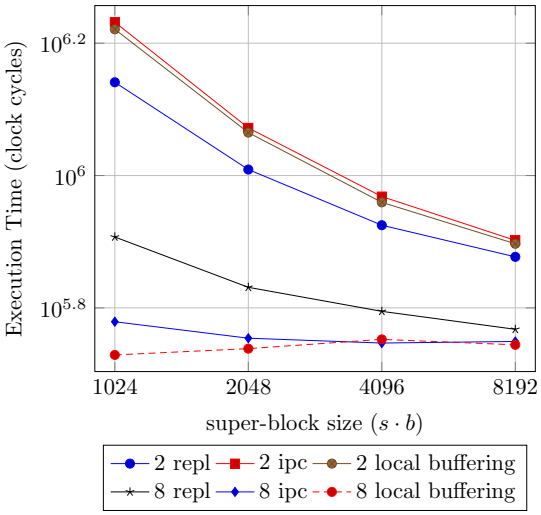
In the transfer regime, performance varies according to the value of k and the number of processors. We can observe in Figure 5.5a that the costs of local buffering and replication are nearly the same, and that replication performs even better for a transfer of block size between 512bytes and 2K. This demonstrates that using DMA for transferring additional data can perform sometimes better than local buffering even for a small value of k, and that keeping shared data in the local store may have a non-negligible cost. Therefore, even when considering contiguous partitioning of data, redundant fetching of shared data using replication strategy can be as efficient, if not more efficient than keeping shared data in the local store. However, the cost of transferring shared data using replication becomes higher than other strategies when the number of processors increases because of the contentions even for small values of k.

In the following, we detail the results for each strategy in terms of efficiency and conformance with the prediction of the models.

- *Replication*: For computation regime, the measured results are very close to the predicted ones. The only source of error would be the contentions on the network with huge network traffic which causes differences in the arrival time of the data. The error between the measured and predicted values is about 3%. In the transfer regime, replication always performs worse than other strategies for 8 processors due to contentions.
- *IPC*: When synchronization is done using messages between neighboring processors, we observe variabilities in the arrival time of data transfers and exchanged messages due to contentions in the network. This effect increases in the transfer regime which makes the gap between the measured and estimated performance larger, with an error of about 6%, that goes to 30% when barriers are used with a high number of processors.
- *Local buffering*: In the computation regime, local buffering outperforms IPC for most of the cases, whereas in the transfer regime, the DMA engine can be more



(a) shared data size 128 bytes



(b) shared data size 1024 bytes

Figure 5.5: Overlapped data in transfer regime

efficient for transferring shared data than copying it from one buffer to another in the local store, despite the inter-processor synchronization overhead.

Note that for a given k, as super-block size increases the cost of transferring shared data relative to the overall execution time decreases and all strategies give similar performance results. In the transfer regime, the gap between estimated and measured performance becomes larger as it is more dependent upon contentions in the network traffic for which our modeling is less accurate. There are two major sources of contentions that we currently do not model:

1. In the 3 stage software pipeline there is an overlap between reading super-block i and writing super-block $i-1$. This is the main reason why the estimated optimal granularity point is in reality still in the transfer regime.
2. Inter-processor synchronization, in which messages exchanged between processors add contention to the network. This overhead increases with the number of processors even for the more efficient inter-processor signaling variant. The exact arrival time of each message is difficult to model due to continuous read/write network traffic and the scheduling policy of the DMA controller.

In the following, we apply the double buffering granularity optimization on a real application working on one-dimensional data: the convolution.

5.3.2.2 Convolution Algorithm

Convolution is one of the basic algorithm is signal processing [Nus81]. Assuming an input signal array X of a large size n and an impulse system response signal array B of a smaller size m ($m << n$), the output system signal array Y of the same size n is computed as follows;

$$Y[i] = \sum_{j=0}^{m} X[i-j] \cdot B[j]$$

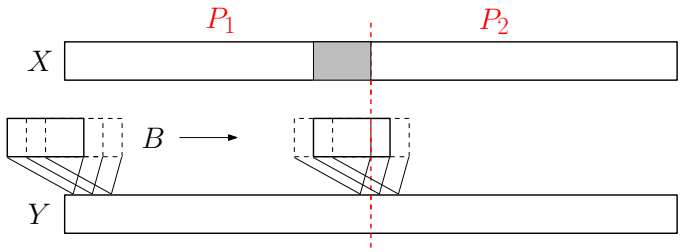


Figure 5.6: Convolution Algorithm.

Therefore to compute each sample $Y[i]$ of the output signal, a window of m data samples is required from input array X. When multiple processors are used, input array is partitioned into contiguous chunks, see Figure 5.6. The area in grey illustrates shared data between processors.

In our experiments, the size of input array X is chosen to be 1Mbytes of data, so it cannot fit in the scratchpad memory, whereas B is small enough to be permanently stored in each SPU's local store. Hence double buffering is implemented to transfer data blocks of array X (resp. Y). As for the synthetic examples, we compare the different strategies of section 4.2 and we vary the size of B to vary the size of shared data.

Signal samples are encoded as *double* data types. The minimal granularity size b is chosen to be the size of the data window required to compute one output data sample, that is $b = m \cdot 8$ (8 being the size of a *double* data type). In the implementation of the algorithm, we use SIMD operations to optimize the code. The measured cost per byte ω is about 53 cycles.

Note that for this algorithm, despite an optimized implementation using vector operations the computation cost per byte ω is much higher than the transfer cost per byte with maximum contentions $\alpha(8)$ (being 7.22 cycles), resulting from the use of the maximal number of available cores. Therefore, the overall execution is always in a *computation regime* for all strategies.

The reason why the cost per byte for this algorithm is so high is the use of *double* data types to encode signals samples which does not fully take advantage of the 16 bytes SIMD engine since operations on *at most* 2 elements of the array can be done in parallel. Floating point data types would take more advantage of the SIMD operations, however at the cost of results accuracy since the Cell B.E. does not support a floating point unit. Besides since SPU's general registers are SIMD registers, this makes operations on the SPU not optimized for scalar operations and branching instructions, resulting in a high execution latency.

Figure 5.7 summarizes performance results for size of $B = 256$ bytes, that is, 32 samples, using 2 and 8 processors. As explained in section 4.2.4, in the computation regime the replication strategy outperforms local buffering and IPC strategies since it avoids the computational overhead at each iteration of copying shared data locally or exchanging data between neighboring processors using synchronous DMA calls. This overhead is proportional to the number of iterations and therefore decreases with higher granularities to be eventually negligible which leads all strategies to perform with nearly the same efficiency.

Moreover, note that in the program execution time estimation, we ignored so far the overhead at each iteration of setup variables which is also proportional to the number of iterations and is hence reduced for high granularity.

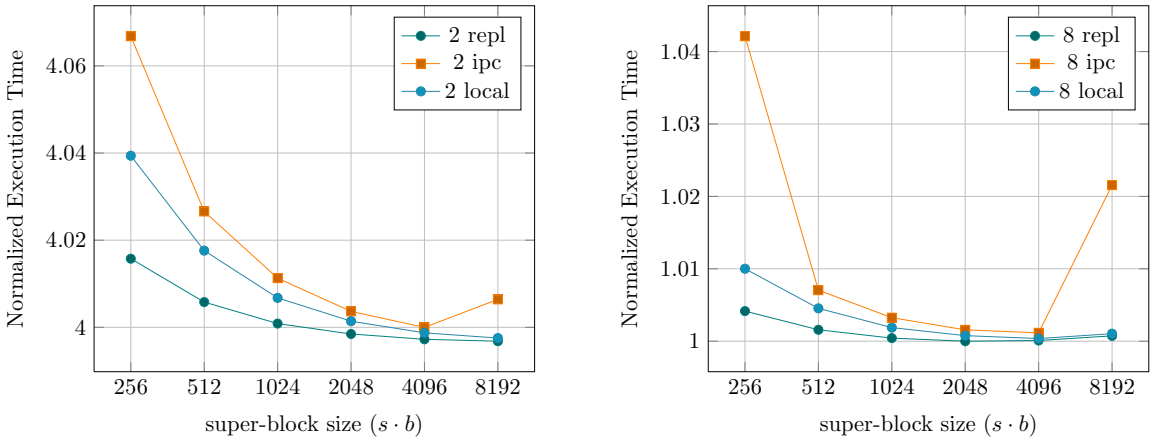


Figure 5.7: Convolution using double buffering, shared data size is 256 bytes

5.3.2.3 Mean Filtering Algorithm

We implemented a *mean filter* algorithm that works on a bitmap image of 512×512 pixels. Each pixel is characterized by its intensity ranging over $0..255$. The output for a pixel is the average of the value of its neighborhood defined as a square (mask) centered around it which corresponds to the neighborhood pattern we considered.

We have experimented with different mask sizes and focus on the presentation of the results for a 9×9 mask, that is, $k = 8$. In order to us SIMD operations to optimize the implementation of the code, we encode a pixel as an integer ($b = 4$ bytes). The computation workload per basic block is roughly $\omega = 62$ cycles.

First we experiment with the influence of the block shape and its implied replicated area on the transfer time discussed in Section 4.3. For this, we fix the quantity $\delta = s_1 \times s_2$ and profile the DMA transfer time for different feasible combinations of (s_1, s_2) so that $s_1 \times s_2 = \delta$. Figure 5.8 plots the DMA transfer time of these shapes for $\delta = 4096$. Note that given the considered mask size, a shape (s_1, s_2) yields a block of $s_1 + 8$ lines, each line corresponding to a contiguous transfer of $b \cdot (s_1 + 8)$ bytes. As argued in section 4.3, the optimal transfer time is obtained neither for square $(64, 64)$ nor the flattest possible $(8, 512)$ super blocks and the best trade-off in this case is $(s_1, s_2) = (32, 128)$.

We then evaluate the effect of the size and shape of the super blocks and the total execution time of the pipeline for different numbers of processors. Fig. 5.9-(a) compares the predicted and measured performance for different block shapes where $s_1 \times s_2 = 1024$ while Fig. 5.9-(b) does the same for $s_1 \times s_2 = 2048$. As one can see, the distance between the predicted and measured values is rather small except for large values of s_1. The major reason for the discrepancy between the model and the reality is that $C(s_1, s_2)$ has non negligible component that depends on s_1 for two reasons. The first is due to the overhead at each computation iteration related to the setting required between the outer loop and the inner loop like adjustment of the pointers for every row, pre-calculation of sums of borders etc. Secondly, the creation of list elements occupies the processor and this overhead is also added to the overall execution time.

Fig. 5.10 combines the measured results for different super block sizes. The measured optimum is obtained for $(4, 256)$ while our calculation yield $(56, 33)$ whose nearest feasible value is $(64, 32)$ whose measured overall performance is less than 10% above the performance for the optimum. The discrepancy can be attributed to the reasons stated above, namely the dependence of C on s_1.

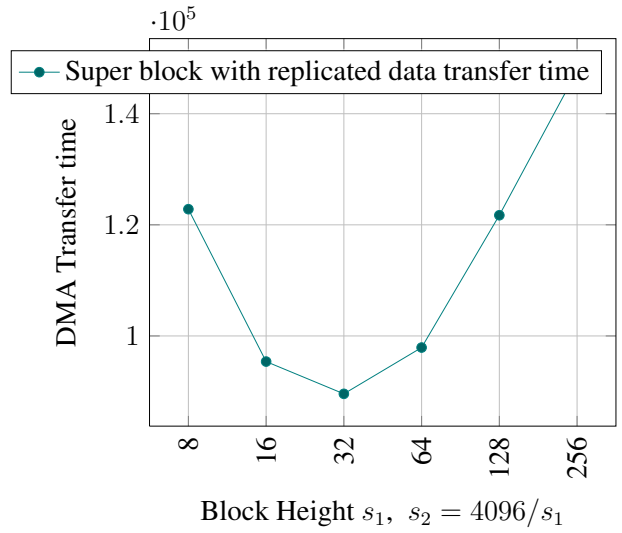


Figure 5.8: Influence of block shape and its replicated data on the transfer time.

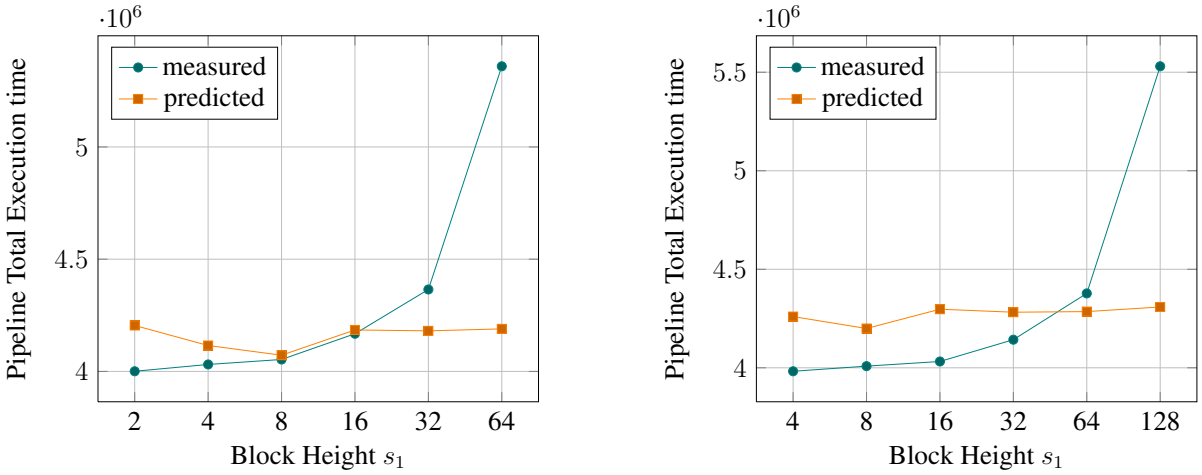


Figure 5.9: Predicted and measured values for different combinations of $s_1 \times s_2$

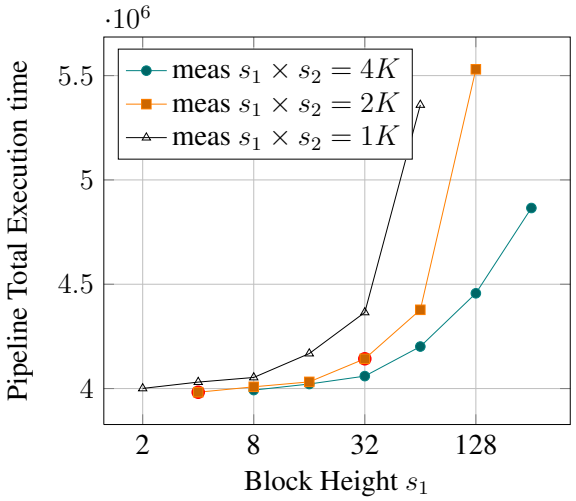


Figure 5.10: Observed optimal granularity $s^* = (4, 256)$ and predicted optimal granularity $s^* = (64, 32)$.

5.4 Conclusion

In order to validate the analysis, we implemented and run double buffering algorithms considering different benchmarks, first synthetic where we vary the computation workload per basic block and the size of shared data, in order to vary the balance between computations and transfers. We considered then real benchmarks where computations share data: a convolution algorithm working on one-dimensional data requiring only contiguous block transfers and a mean filtering algorithm working on two-dimensional data requiring rectangular block transfers.

Overall the analytical results are close to the measured ones which proves that we captured the important features of the problem. However, some discrepancy remains mainly because of the following reasons:

- Variabilities in the architecture which influence in particular the value of α and where an average value is clearly not enough to capture these variabilities.
- We ignored so far the overhead per line involved in the computation time of a rectangular block. It turns out that in the Cell B.E. architecture, this overhead is significant since, i) the issue of each list element is performed by the software which adds to the processing time and ii) the Cell B.E. processors are vector processors not optimized for scalar and branching instructions which makes the extra processing required for the computation of a rectangular block not negligible.

We believe that a right choice of the parameters values along with more detailed models can further improve granularity and performance predictions.

Conclusions and Perspectives

The critical path of this thesis was to understand the architectural context and the new challenges emerging from the current trend for designing embedded multi-core systems, which are in some aspects very different from general-purpose multi-core platforms. In fact, a particular difficulty in the embedded multi-core domain is the cultural gap between the software and hardware communities that hold different partial views of the same system and thus problems.

For such platforms, efficient use of the memory hierarchy is crucial for performance and constitutes one of the major new challenges. It breaks with the implicit assumption of an unlimited memory space available to the program and thus puts a heavy burden on the software and the programmer who have to manage data movement in the memory hierarchy, granularity choice, synchronization with the computations, etc.

To tackle this problem, we have targeted a relatively-easy, but important, class of applications with *regular patterns* of computation and *high volume* data transfers. This choice may look too restrictive, however array processing algorithm constitute a large part of today's embedded applications and their efficient software implementation on multi-core architectures is an activity that will still occupy a lot of programmers in the coming future.

Under some simplifying assumptions we constructed models that capture the main features/parameters of the problem, that is the logical description of the applications and the DMA specification, with a clear separation between parameters characterizing the applications and those which are specific to the hardware platform. A crucial point of our methodology is that the hardware characterization and modeling phase can, in principle, be done once for each new platform and then be used by different applications running on it.

The analysis turned out to approximate reasonably well the behavior of implemented benchmarks on a real architecture. However, we are of course aware of the fact that each real program and each architecture will have its own particularity, more complex and richer in parameters than the model we have built. The work presented in this thesis is just a first step and our main concern was to provide an abstract view of the problem towards a more systematic way for solving such problems than the actual pure trial and error engineering method.

There are many possible extensions for this work and a lot is still to be done, including,

– Integrate different/more complex features of data parallel applications such as different data sharing patterns, potentially involving temporal dependency and thus allowing data dependent execution;
– Capture variations which is necessary if we want to move a step towards more realistic results. Sources of variations can be software and/or hardware. In software they appear for instance naturally in some filtering algorithms where the degree

of filtering and thus computations varies according to the structures of the image. Hardware variations are, as discussed throughout this thesis, mainly due to the contentions on the NoC and the main memory side. We think that a more detailed model of DRAM features would help improve granularity and performance predictions and that a dynamic tuning of the granularity between subsequent iterations maybe required to adapt to the variations in the memory access latency;

– In contrast with data-parallelism, task-level parallelism usually use multithreading to hide memory latency where multiple threads run on the same core and a context switch occurs at the request of data that is not available in local memory. Combining both data and task parallelism is in practice required since different applications can run simultaneously on the same multi-core fabric. This will change the symmetric nature of the problem and may create new bottlenecks. Consideration of code distribution should also be taken into account;
– Adapt our analysis to a distributed DMA architecture where data transfers can be scheduled at the super block rather than the packet level. This way, useless contentions between sub tasks of the *same* application can be avoided;
– So far, we considered only two memory levels in the hierarchy, main memory and processors local memory. We can also consider a third memory level (which is in practice already available in platforms such as P2012). More generally, the memory hierarchy can be viewed as a tree where each node features a different speed and capacity, and one (or multiple) DMA (s) engines are used to transfer data from one memory location to another;
– Finally, we think that the best way to leverage this work is to integrate it in a complete compilation flow where the programmer writes his program as a sequential data parallel loop and a double buffering algorithm source code is generated automatically with the appropriate partitioning for the target platform, that varies as we vary the hardware;

We are currently extending our work to P2012, which is the initial motivation for this thesis. The problem as it is so far described already matches the inter-cluster view of P2012 where each cluster has its own local memory and DMA engine. In the same cluster, the programmer can chose between two strategies of moving data, i) a “liberal” approach where each core copies data independently of other cores in the cluster thus issuing a different DMA request per core, data is however stored in the shared memory. ii) a “collaborative” approach where cores in the same cluster issue simultaneously one common transfer request to serve their respective computations. It will be interesting to compare both approaches as the first offers more synchronization flexibility and the second matches the hardware view.

Bibliography

[AB86] James Archibald and Jean-Loup Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4):273–298, September 1986.

[ADC+03] A. Artieri, V. D'Alto, R. Chesson, M. Hopkins, and M. C. Rossi. Nomadik - Open multimedia platform for next generation mobile devices. Technical report, 2003.

[AKN95] Anant Agarwal, David Kranz, and Venkat Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared memory multiprocessors. *IEEE Trans. Parallel Distributed Systems*, 6:943–962, 1995.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[AP01] Turgay Altilar and Yakup Paker. Minimum overhead data partitioning algorithms for parallel video processing. In *Proceedings Domain Decomposition Methods Conference*, pages 25125–8, 2001.

[BM02] Luca Benini and Giovanni De Micheli. Networks on chips: A new soc paradigm. *Computer*, 35:70–78, 2002.

[BSL+02] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, CODES '02, pages 73–78, New York, NY, USA, 2002. ACM.

[BZZL08] Shuwei Bai, Qingguo Zhou, Rui Zhou, and Lian Li. Barrier synchronization for cell multi-processor architecture. In *Ubi-Media Computing, 2008 First IEEE International Conference on*, pages 155 –158, august 2008.

[CB94] T.-F. Chen and J.-L. Baer. A performance study of software and hardware data prefetching schemes. *SIGARCH Comput. Archit. News*, 22:223–232, April 1994.

[CKP91] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ASPLOS-IV, pages 40–52, New York, NY, USA, 1991. ACM.

[CMLS11] Scott Cotton, Oded Maler, Julien Legriel, and Selma Saidi. Multi-criteria optimization for mapping programs to multi-processors. In *SIES*, pages 9–17, 2011.

[DDS95] Fredrik Dahlgren, Michel Dubois, and Per Stenström. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 6:733–746, July 1995.

[DRV00] Alain Darte, Yves Robert, and Frederic Vivien. *Scheduling and Automatic Parallelization*. Birkhauser Boston, 1st edition, 2000.

[Ehr00] M. Ehrgott. *Multicriteria optimization*. Lecture Notes in Economics and Mathematical Systems. Springer-Verlag, 2000.

[fClB95] Tien fu Chen and Jean loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44:609–623, 1995.

[FHK+06] K. Fatahalian, D.R. Horn, T.J. Knight, L. Leem, M. Houston, J.Y. Park, M. Erez, M. Ren, A. Aiken, W.J. Dally, et al. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 83–es. ACM, 2006.

[Fri02] J. Fritts. Multi-level memory prefetching for media and stream processing. In *Multimedia and Expo, 2002. ICME '02. Proceedings. 2002 IEEE International Conference on*, volume 2, pages 101–104 vol.2, 2002.

[Gro08] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[Gsc07] M. Gschwind. The cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35(3):233–262, 2007.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[IBM08] IBM. Cell SDK 3.1. `https://www.ibm.com/developerworks/power/cell/`, 2008.

[IBM09] IBM. Cell Simulator. `http://www.alphaworks.ibm.com/tech/cellsystemsim`, June 2009.

[IM02] Anoop Iyer and Diana Marculescu. Power and performance evaluation of globally asynchronous locally synchronous processors. *Computer Architecture, International Symposium on*, 0:0158, 2002.

[kLH95] Chi kin Lee and Mounir Hamdi. Parallel image processing applications on a network of workstations. *Parallel Computing*, 21(1):137 – 160, 1995.

[KPP06] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *Micro, IEEE*, 26(3):10 –23, may-june 2006.

[KYM+07] Timothy J. Knight, Ji Young, Park Manman, Ren Mike Houston, Mattan Erez, Kayvon Fatahalian, Alex Aiken, William J. Dally, and Pat Hanrahan. Compilation for explicitly managed memory hierarchies. In *In PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 226–236. ACM Press, 2007.

[LCM11] Julien Legriel, Scott Cotton, and Oded Maler. On universal search strategies for multi-criteria optimization using weighted sums. In *IEEE Congress on Evolutionary Computation*, pages 2351–2358, 2011.

[LGCM10] Julien Legriel, Colas Le Guernic, Scott Cotton, and Oded Maler. Approximating the pareto front of multi-criteria optimization problems. In *TACAS*, pages 69–83, 2010.

[LLG+90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. *SIGARCH Comput. Archit. News*, 18(3a):148–159, May 1990.

[LM87] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[LPB06] Mirko Loghi, Massimo Poncino, and Luca Benini. Cache coherence tradeoffs in shared-memory mpsocs. *ACM Trans. Embed. Comput. Syst.*, 5:383–407, May 2006.

[MB09] Andrea Marongiu and Luca Benini. Efficient openmp support and extensions for mpsocs with explicitly managed memory hierarchy. In *DATE*, pages 809–814, 2009.

[MBB10] Andrea Marongiu, Paolo Burgio, and Luca Benini. Evaluating openmp support costs on mpsocs. In *DSD*, pages 191–198, 2010.

[MCT96] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, July 1996.

[MG91] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12:87–106, 1991.

[Nus81] Henri J. Nussbaumer. *Fast Fourier transform and convolution algorithms*. Springer-Verlag, Berlin ; New York :, 1981.

[OHL+08] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879 –899, may 2008.

[Ope08] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, 3.0 edition, May 2008.

[SBKD06] José Carlos Sancho, Kevin J. Barker, Darren J. Kerbyson, and Kei Davis. Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[SC10] STMicroelectronics and CEA. Platform 2012: a many core programmable accelerator for ultra efficient embedded computing in nanometer technology, 2010.

[SK08] J.C. Sancho and D.J. Kerbyson. Analysis of double buffering on two different multicore architectures: Quad-core Opteron and the Cell-BE. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

[SNBS09] C.D. Sudheer, T. Nagaraju, P.K. Baruah, and Ashok Srinivasan. Optimizing assignment of threads to spes on the cell be processor. *Parallel and Distributed Processing Symposium, International*, 0:1–8, 2009.

[Ste90] Per Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, June 1990.

[SYN09] S. Schneider, J.S. Yeom, and D.S. Nikolopoulos. Programming multiprocessors with explicitly managed memory hierarchies. *Computer*, 42(12):28–34, 2009.

[TKA02] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, UK, 2002. Springer-Verlag.

[WBM+03] Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems. Guided region prefetching: a cooperative hardware/software approach. *SIGARCH Comput. Archit. News*, 31:388–398, May 2003.

[WJM08] W. Wolf, A.A. Jerraya, and G. Martin. Multiprocessor system-on-chip (MPSoC) technology. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(10):1701–1713, 2008.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *SIGPLAN Not.*, 26(6):30–44, May 1991.

[WM95] Wm. A. Wulf and Sally A. Mckee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23:20–24, 1995.

[YRL+09] S.S.J.S. Yeom, B. Rose, J.C. Linford, A. Sandu, and D.S. Nikolopoulos. A Comparison of Programming Models for Multiprocessors with Explicitly Managed Memory Hierarchies. 2009.

[ZM02] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In *CADE*, pages 295–313, 2002.

MOT-CLEFS

mot-clef1, mot-clef2, mot-clef3

TITLE

Optimizing DMA Data Transfers for Embedded Multi-Cores

KEYWORDS

keyword1, keyword2, keyword3

ADRR : Adresse Labo

ISBN : □□□□□□□□□□□□□□□